LLMs PROGRAM USING REUSABLE AND COMPOSABLE ABSTRACT REPRESENTATIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

When writing code, the same computation can be expressed in different programming languages and through different implementation strategies. We show that the activations of LLMs capture this shared structure in reusable and composable abstract representations when they generate code. These representations emerge in the middle layers, are reusable across programming languages, and afford the model an impressive amount of generalization ability when generating code under various interventions. Using activation steering, we show that function, programming language, and implementation strategy can be independently manipulated. The independent interventions on language and function can be flexibly composed to control code generation, even in cases when the prompt contains no information about language and function. We additionally find that finetuning on examples of arbitrarily defined functions in one programming language transfers to other languages, consistent with the models reusing abstract features of code generation that are shared across different examples. Finally, to study how such abstract representations come to be, we implement a minimal model of shared, reusable, and composable structure in a synthetic setting, and show that transformers trained on such data learn representations that reuse these components across contexts.

1 INTRODUCTION

Programming repeatedly combines familiar computations with different languages and implementation choices. A graph search, for example, can recur across programs with different syntax, control flow, and data structures. A competent programmer can recognize what these implementations share, and reason about the abstract computation, separately from the details of its realization. This allows knowledge acquired in one setting to support programming in another; understanding a graph-search algorithm in Python can help a programmer implement it in Java, even when the implementation requires different language constructs and data structures. Rather than learning every realization independently, programmers can reuse their understanding of the computation while adapting its implementation to the setting. In this work, we show that coding models also learn such abstractions from the shared computational structure of program data. These representations are built from reusable contributions for what a program computes, the language in which it is expressed, and its implementation strategy.

Our work builds on the Factored World Hypothesis (FWH), which proposes that predictive models can represent independent components of a data-generating process in separable latent state spaces (Shai et al., 2026). Here, we extend the theoretical framework to training data from multiple sources that share computational parts (Ray et al., 2026). When training data from different sources contain these shared mechanisms, we anticipate the emergence of reusable factored representations. That is, representations that track information about the shared mechanism separately from information about the source-specific components. Representations that take this form are more dimensionally efficient at no predictive cost, because the joint representation can be fully recovered from the two marginals: one over the shared mechanism and one over the remaining mixture.

In the following, we ask whether a model's activation state Z (in a transformer (Vaswani et al., 2017), the residual stream, in a recurrent neural network, the hidden state) learns to exploit this kind of shared structure. We test reuse directly in open-source coding models' internal representations. To begin, we find that when a programming instruction is put into context, the most performant models

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.

retain an internal state that supports code generation, even after direct access to the specification is removed. This compiler-like representation gives us an internal target of study.

Through this compiled state, we ask whether the model represents a function it's been asked to implement separately from the programming language in which it's been asked to write. Our contributions are as follows:

1. **Coding LLMs create a compiled program state representation.** Across 26 instruction-tuned models, the strongest coders keep enough task information at a shared suffix to write correct code after the specification is masked (Section 3.1).
2. **LLMs represent function, language, and strategy as reusable, composable components.** Interventions estimated only from other languages transport, install, and compose function, language, and strategy in three LLMs and a modern RNN (Section 4.1).
3. **Training on data with shared mechanisms produces reusable representations.** In a controlled synthetic setting, next-token prediction on data whose sources share a mechanism is enough to produce reusable, composable structure (Section 4.2).
4. **Reusable abstractions let models generalize across languages.** State from a single in-context example transfers across languages, and Python-only fine-tuning on invented algorithms transfers to held-out languages (Section 5).

2 Related Work

Shared representations and Cross-Language Generalization Prior work has found semantic structure in code-model representations, including emergent program semantics (Jin & Rinard, 2024), language-agnostic code embeddings (Utpala et al., 2024), and shared conceptual representations across natural languages (Dumas et al., 2025). Other work directly encourages abstraction, for example, by training models against compiler intermediate representations (Paul et al., 2024). Similarly, cross-programming-language transfer is well established behaviorally. Language models often transfer knowledge between languages, with reliability depending on the source language, target language, and task (Baltaji et al., 2025). Parallel-SFT shows that explicitly aligning equivalent implementations across languages can improve zero-shot transfer during subsequent code training (Wu et al., 2026), while other work studies what syntax and semantics transfer when models encounter unfamiliar programming languages (Kumar et al., 2026). We directly extend this work with an in-depth representational study, asking if the representations naturally learned during pretraining are factored abstractions and show that behavior like cross-language transfer may be a natural consequence of a factored, reusable representation.

Task and function representations. Task vectors can summarize demonstrations (Hendel et al., 2023), while function vectors localized to attention-head activity can causally induce learned input-output mappings and exhibit limited compositionality (Todd et al., 2024). In-context vectors provide a related latent-space intervention framework (Liu et al., 2024). Recent work has further asked whether these representations are invariant to how a task is presented. Davidson et al. (2025) find that instructions and demonstrations rely largely on distinct, partially overlapping function-vector mechanisms, whereas Luo et al. (2025) find task representations that transfer across modality and presentation format. Here, we use representations obtained from language models that correspond directly to programming languages, functions, and implementation details, and show that these can in general be transferred and composed to induce new computation.

Activation and cache interventions. Latent steering vectors, Activation Addition, and Contrastive Activation Addition manipulate generation using activation-space directions derived from desired behaviors or contrastive prompts (Subramani et al., 2022; Turner et al., 2023; Rimsky et al., 2024); instruction vectors can similarly be composed to control multiple behavioral constraints (Stolfo et al., 2025). Code-specific work has used activation steering for properties such as types, language, and library choice (Lucchetti & Guha, 2025; Rahman et al., 2026). Prefix-tuning and KV-cache steering demonstrate that persistent prompt or cache state can also exert substantial control over subsequent generation (Li & Liang, 2021; Belitsky et al., 2025). In our work we use steering as our intervention of choice to causally test the predictions a factored, reusable representation makes.

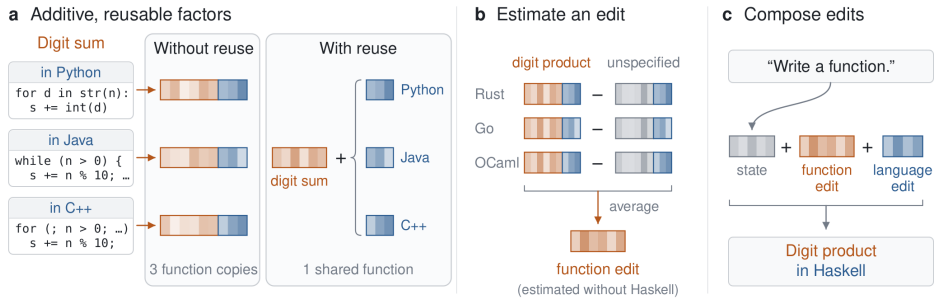

Figure 1: **(a)**: The factored world hypothesis motivates additive activation states; when multiple factored sources reuse the same mechanism, a reusable representation avoids redundancy. **(b,c)**: Testing composability and reusability of coding representations. In (b), we find a steering vector for a fixed function without ever asking the model to output Haskell code (separately, we also find a steering vector for Haskell, without ever asking the model to implement the target function.) In (c) we use both steers at once to implement the target function in Haskell.

Factoring. An intuition has been building that neural networks often seem to factor low-level syntax from abstract semantics. This is seen in image generation, where the style of a painting can be altered independently of its content (Gatys et al., 2015). And it's seen in language models, where syntax and semantics factor both at the level of particular words (Feucht et al., 2025) and at the level of full sentences (Acevedo et al., 2026).

Our work lends further credence to this intuition, while providing a very sharp set of examples via compositional programming representations, and supplying a toy model that helps explain where this factored abstract structure can come from.

3 Factored Program State and Causal Tests

Programming tasks provide a natural example of a structured, non-ergodic data source (Ray et al., 2026). Computation recurs across many otherwise different contexts. For example, a graph search can be implemented in Python or C++, recursively or iteratively, and using depth-first or breadth-first traversal. We model these recurring structures with task variables $\theta = (f, \lambda, s)$, where f denotes the computation or input–output mapping, λ the programming language, and s the implementation strategy. We can think of a program u as being drawn from a task-conditioned source.

$$u \sim p(u \mid \theta). \tag{1}$$

Conditioning on a task θ means supplying a prompt that specifies f, λ, and s. Language can take different values, such as "Python" or "Java", while keeping one or both of function and implementation fixed, and likewise for the other variables.

The Factored World Hypothesis (Shai et al., 2026) motivates a decomposition of some persistent *activation state* $Z(\theta)$ into separable, additive contributions associated with function, language, and strategy. We denote these contributions by $Z_F(\theta), Z_L(\theta), Z_S(\theta)$.

Our *reuse hypothesis* further proposes that each contribution depends primarily on its own task variable. For language, we express this as $Z_L(\theta) \approx Z_L(\lambda)$: the same language representation should be shared across functions and strategies. Applying the reuse assumption to function and strategy gives Z the form

$$Z(\theta) \approx Z_0 + Z_F(f) + Z_L(\lambda) + Z_S(s). \tag{2}$$

Here Z_0 denotes whatever part of the activation state is shared by all tasks. Some decomposition of this form is predicted to hold for each task, though it may be subject to compatibility constraints that restrict which combinations can occur.

We test the operational prediction that *individual factor changes* transfer across dissimilar contexts: a state change intervention estimated in one context should produce the corresponding change in another. For example, an edit estimated from analogous tasks in non-Python languages should be

able to change the state for summing digits in Python into one that produces their product in Python. This prediction follows directly from equation 2, because

$$Z(f^\star, \lambda, s) - Z(f, \lambda, s) \approx Z_F(f^*) - Z_F(f). \tag{3}$$

The final displacement vector depends on f, not the source language or implementation strategy. Here, the activation state need not equal a Bayesian belief or inherit tensor-product coordinates. The hypothesis we test follows from, but does not require orthogonal subspaces. Nor does it depend on the pressure to minimize representational dimension. In fact, it does not even require that the factoring holds at every layer of a model.

Testing this prediction does require identifying a reliable activation state $Z(\theta)$ through which the model can represent, and we can modify, the task specification. In the next section, we empirically find a candidate site to hold such abstractions in LLMs. Then, we use it to investigate causal signatures of representational reuse using natural language programming prompts. We then show the same behavior in a toy process with an exactly known data distribution in which theory predicts such a representation.

3.1 Modern Models Exhibit Compiler-Like Prompt State

We look for an intervention site by asking whether a model retains useful task information after direct access to the program specification tokens is removed. Each prompt contains a task-bearing prefix followed by a shared suffix. During prefill, we process the full prompt and cache the key–value states. We then mask attention to the specification positions, denoted $\mathcal{D}$, and generate a program. This tests whether information propagated beyond the specification token positions can support the task; prompt details are in Appendix A.4.

We screen 26 instruct-tuned LLMs on 310 coding tasks spanning 10 languages, then repeat the task masking out attention to the prefix. Success requires passing all unit tests while only attending to the shared suffix. (Appendices A.5 and A.6).

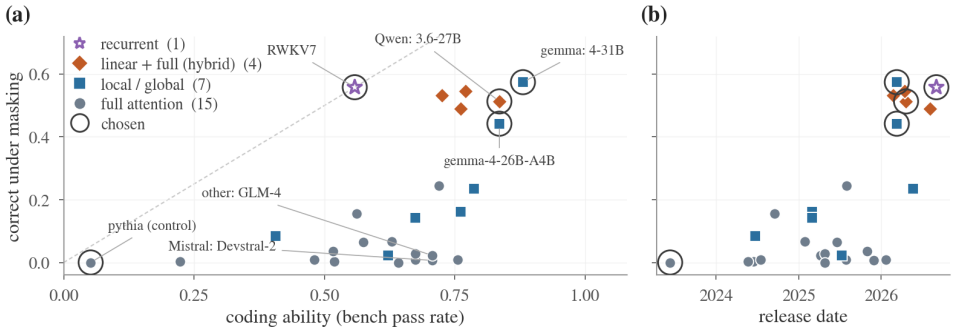

Figure 2: **(a)**: The most capable coding models are more resistant to not being allowed to attend to the description after prefill. These models also mix local or linear attention with global, or are fully recurrent. **(b)**: This also largely correlates with the recency of the model.

The most capable coding models retain substantial performance under this intervention, indicating that the task information provided in the prefix is "compiled" into a representation that exists in the activations at the shared suffix context positions. (Fig. 2). This provides a controllable local site that we use as our $Z(\theta)$ for testing whether the function and language contributions can be reused separately. We now proceed to test if this $Z(\theta)$ behaves as predicted in Eq. 2.

3.2 Intervention Design

To compare states across tasks, we build all prompts using the same format. For task θ, the full prompt is

$$x = d_\theta \,\|\, \sigma, \tag{4}$$

where d_θ is a task-bearing prefix, σ a shared suffix, and $\|$ token concatenation. d_θ is constructed to ensure that all prompts reach the suffix at the same token positions, so differences in the activations

induced by the suffix context positions reflect only information propagated from the prefix. We call prompts used to estimate an edit *donors* and the prompt whose state receives the edit the *recipient*. Natural language prompt construction details and examples can be found in Appendix A.4.

After prefill, the editable key–value cache block is $Z(d_\theta) = \{K_{\ell,i}, V_{\ell,i}\}_{\ell\in\mathcal{L},\, i\in\mathcal{S}}$ where $K_{\ell,i}$ and $V_{\ell,i}$ are the key and value tensors at layer ℓ and position i; $\mathcal{L}$ selects layers and $\mathcal{S}$ suffix positions. In each intervention, we apply $\widetilde{Z} = Z + \alpha\Delta$, with displacement Δ and dose α, matching each layer, position, and key or value entry. Description-access removal is a separate operation, specified for each experiment– but effectively amounts to masking the attention mechanism so that it may only attend to the shared, intervened suffix representations formed while the description was visible.

Under the reuse hypothesis, contrasting two functions in the same language cancels the language contribution. Write $\overline{Z}(\theta)$ for the mean suffix state over prompts that realize θ. An edit from source function f to target $f^\star$ is estimated by

$$\Delta_F[f^\star \leftarrow f] = \underset{\lambda'\in\Lambda}{\text{mean}} \left[\overline{Z}(f^\star, \lambda') - \overline{Z}(f, \lambda')\right]. \tag{5}$$

Here Λ is the set of languages whose prompts serve as donors. When Λ leaves out the recipient's language, the edit is built entirely from other languages, so success shows that the factor transfers across contexts; we call such a test a "leave-one-context-out" intervention. When Λ contains only the recipient's language, we call it a "same-context-only" intervention. A language edit, Δ_L, analogously averages contrasts across various functions.

Three tests of reuse. We will test increasingly demanding uses of the same estimator. Below, ? indicates a task prompt where one or more of the factors are left unspecified.

- **Transport:** change (f, λ) to $(f^\star, \lambda)$ using $\Delta_F[f^\star \leftarrow f]$, preserving the recipient language.
- **Installation:** start from $(?, \lambda)$ and supply the target function using $\Delta_F[f^\star \leftarrow ?]$.
- **Composition:** start from $(?, ?)$ and add separately estimated function and language installation edits to obtain $(f^\star, \lambda^\star)$.

Below, we report the fraction of recipients that pass tests designed for the target prompt. Appendix A.1 gives the full estimator and intervention equations. Each experiment below defines the specific ways in which this overarching approach applies.

4 Testing Transport, Install, and Compose

4.1 Language Models Reuse Function, Language, and Strategy

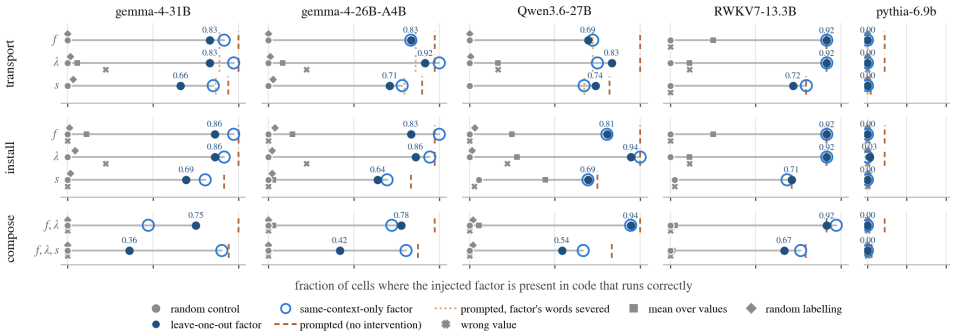

Figure 3: **Function, language, and strategy support causal reuse.** For each condition, we report the fraction of evaluation cells in which the generated program both runs correctly and exhibits the injected factor. Function and language factors estimated outside the recipient context generally recover most of the effect of same-context-only interventions, and strategy factors likewise causally control recursive versus iterative realization. Function–language composition remains strong, while simultaneous composition of function, language, and strategy is substantially more difficult. The Pythia control shows no corresponding causal reuse. Interventions with s have 72 trials, all others have 36 (since Python is never a recipient).

From the retention screen in Section 3.1, we select the top three models, `Gemma-4-31B`, `Gemma-4-26B-A4B` (Team et al., 2026), and `Qwen3.6-27B` (Bai et al., 2023). This also gives us a regular local/global attention transformer, a Mixture-of-Experts (MoE), and a hybrid model with linear and regular attention. We carry `Pythia-6.9b` (Biderman et al., 2023) through the rest of the experiments as a control, since it has not been extensively trained on code, it lets us validate that our manipulations do not trivially work in any language model. Additionally, we perform all interventions on `RWKV7-13.3B` (Peng et al., 2025), due to its recurrent state giving us an exact place to intervene since there is no question of where its representation exists.

We apply the transport, installation, and composition tests of Section 3.2 to a subset of 4 tasks the models *are* able to solve without access to the description during generation in 10 different programming languages. For transport and installation, after prefill, we mask attention to the description tokens, forcing the model to generate entirely based on its compiled state. Qwen's recurrent memory also receives the donor-derived edit. Appendix A.4 and Appendix A.6 contain full intervention details and choice of tasks and languages.

Strategy extends the same estimator to contrasts between implementations, averaged over other function–language contexts. In our case, we define strategy to be the choice between iteration or recursion. We test function–language composition and the harder combination of all three factors. We consider an intervention a success when the decoded program passes 6 unit tests. Note that we always leave out `Python` as a recipient since this is every tested model's default language and would inflate our score.

Figure 3 shows that the causal predictions of factorization hold across the tested models. Filled blue markers use leave-one-context-out estimates; open blue markers use same-context-only estimates. Gray markers show control interventions, and dashed orange lines show ordinary prompting without intervention. Together, these results show that function, language, and strategy are causally reusable components of the compiled program state. In our design, the donor and recipient share exact wording of each description, which likely contributes to the high success rate. In Appendix A.16 we show additional results where donors and recipients are paraphrases of one another.

4.2 How could this be happening?

What structure in training data makes this kind of shared representation useful? We propose that it is data in which many sources reuse a common mechanism, like how programs in different languages reuse the same computations. In this section, we show why this type of data favors a representation that stores the shared mechanism once. We then build a small programming-inspired dataset with exactly this structure, train a transformer on its output alone, and apply the transport, installation, and composition tests of Section 4.1. If next-token prediction alone is enough to produce a reusable factored state, the trained model should show the same causal reuse as the language models in Fig. 3. Indeed this is what we find.

Consider data from many sources that share a mechanism, such as the same function (say, x^2) written in different programming languages (say, Python or C). A network can represent the shared mechanism separately inside each source, or once for all of them (Fig. 1a). Both support the same next-token predictions, but representing it once needs fewer dimensions, and the saving grows linearly with the number of sources (with K sources, representing it once saves $K - 1$ copies of the shared mechanism).[1]

Our toy dataset then has the following structure: each sequence comes from one of three *functions* (the analogue of x^2), expressed in one of eight *languages* (the analogue of Python or C): the function sets the statistics of the symbols, and the language sets which tokens carry them. The same three functions appear in every language, and the transformer sees only the resulting tokens (Fig. 4a; Appendix A.3). Languages also insert runs of blank tokens that carry no task information, which we later use as the shared suffix.

We train transformers with both an eight-position attention window and full dense attention. Both learn the task similarly well ($\approx 8 \times 10^{-4}$ nats from the optimal loss, averaged over three seeds). The windowed model must relay task information as earlier positions become inaccessible, which

[1] This relies on the FWH result (Shai et al., 2026) that networks can represent a product of factors as a direct sum of their marginals. See Appendix A.2 for more formal detail.

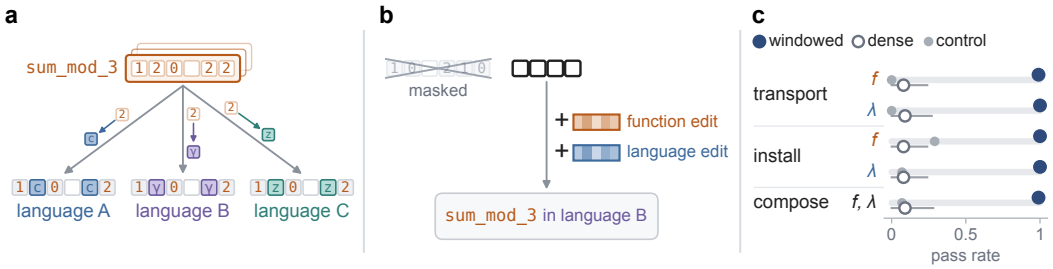

Figure 4: **A toy training dataset with a shared mechanism.** **(a)** A function (here `sum_mod_3`, one of three) generates a string of symbols (orange). A language turns that string into tokens: each symbol is usually written with a token shared by all languages (gray) and sometimes with a token of that language's own (colored; language A writes 2 as c, language B as γ). The function is a shared mechanism, analogous to how digit sum is shared by its Python and Rust implementations. The transformer is trained on the tokens alone. **(b)** The composition test of Section 3.2: from a prompt that specifies neither factor, we mask the description and add, at the blank suffix, a function edit estimated only from other languages and a language edit estimated only from other functions. **(c)** With an eight-token attention window, the model passes transport, installation and composition in nearly every trial (compose: 0.99). A dense-attention model with the same loss does not (0.09): even its own prompt, once masked, leaves little at the suffix (0.08), so accurate prediction alone does not guarantee a reusable state. Leave-one-context-out edits, three seeds (marker: mean; bar: seed range); control is the stronger of the random-direction and wrong-value edits. Full results in Fig. 8.

creates an incentive for the model to "compile" the prompt state in the way we observe in the more competent LLM coders.

Here, d_θ is a token sequence of 36 tokens whose statistics provide evidence about the function and whose tokens indicate the language. A $(?, \lambda)$ prompt omits function information by using a function-ambiguous sequence (one that leaves the optimal observer near a uniform guess), while a $(f, ?)$ prompt leaves language ambiguous by using only shared tokens; a $(?, ?)$ prompt uses both at once. Here, function specified recipients are selected so that they reveal their task to an optimal Bayesian observer with a certainty of $> 95\%$. A series of four "blank" tokens forms a suffix shared by all tested contexts, and attention to the 36 token prefix is masked during generation in each trial (Fig. 4b).

Because we have an optimal ground-truth distribution of tokens for each prompt, we can determine the success of our interventions by querying the model's next token distribution at just a few positions past the shared suffix. Language, for example, can be read from the probabilities assigned to language-specific tokens immediately after the suffix. Additional training details, prompt construction, filtering, donor counts, and scoring rules are in Appendix A.3.

Figure 4c (full results in Fig. 8, Appendix A.3.1) shows the transport, installation, and composition tests work nearly perfectly for all recipients and donors in the windowed attention model. We see that these interventions work equally well whether they use same-context-only or leave-one-out steering– supporting the claim that this toy model has learned composable, reusable representations for language and function.

The dense attention model achieves similarly low prediction loss but displays much weaker function reuse at the suffix positions. We note that the failure of these interventions merely shows that there is not a reusable representation *at the shared suffix*– not that the dense models do not have a reusable representation. This is consistent with the model's poor performance under prompt masking, even when given the correct target prompt. Additional controls and model architectures are described in (Appendix A.3.1).

5 Abstractions Support Generalization

The preceding experiments show that a specified computation can be realized in different programming languages through a reusable internal state. But the value of an abstraction is not only that

it supports different realizations: it should also allow information acquired from one realization to support another. We therefore ask whether reading a single program produces a computational state that can be reused beyond the example's language. We present one implementation, harvest the persistent state accumulated at a shared suffix, and transplant that state into an otherwise example-free prompt. We pick 2 regular algorithms and 2 "nonce" algorithms (arbitrary sequences of steps that do not have a "common" name) for each model that exhibits the best performance when attention to the example is masked. Crucially, the target programming language is specified only after the state has been formed, separating acquisition of the computation from the choice of its subsequent realization. We intervene only with programs that successfully survive masking as in section 3.2.

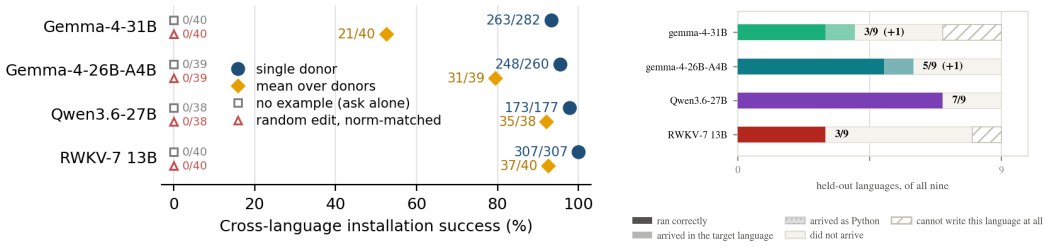

Figure 5: **(a)**: A state harvested from the suffix prompt with a single example in one language transplanted into an example-free prompt with a different language specified produces a correct program in the new language. **(b)**: Models trained on a single algorithm in Python learn that algorithm in several held-out languages.

Figure 5a shows that states transplanted from a single prompt with a single language generalize broadly to all other languages. What the model stores at the suffix is thus sufficient for rendering the algorithm in far more languages than it has seen it in. Finally, we try an extreme version of generalization: fine-tune existing models on just a single unseen algorithm in Python to see if they generalize to held-out languages. Wu et al. (2026) already showed that training on multiple languages in parallel improves held-out language performance. However, if there is truly a shared representation, a single example should in theory generalize. Figure 5b shows the results for an invented algorithm `vantril`. Even in such a data-starved regime, the model manages to generalize to unseen languages. We note that this generalization can likely be considerably improved with more fine-tuning, but consider this beyond the scope of this paper. We expand on these results and describe the full algorithms and experiments in Appendix A.7 and A.8.

5.1 What is an Abstraction?

We now turn to an examination of what is actually contained in our $Z(\theta)$ state. Previous experiments could succeed if the model simply stored a summary of the prompt or, in the in-context learning case, the demonstration it saw in its KV-cache. To address this, we look at interventions where the donor is not averaged but comes from a single language.

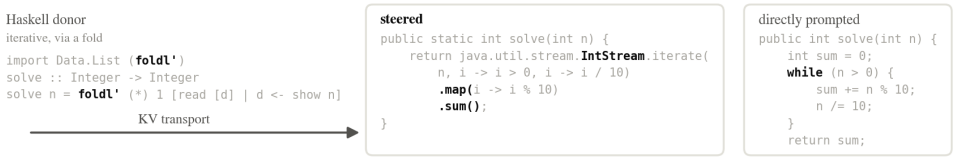

Figure 6: **Cross-language strategy transport preserves implementation abstractions, not only coarse strategy labels.** Gemma steered with a fold-based iterative donor from Haskell induces a Java-native reduction/stream implementation rather than either Haskell surface syntax or Java's directly prompted while-loop. This reduction-like Java implementation is not produced when the recipient is directly asked for iteration.

For strategy, in most cells, the transported state behaves like a label: when it changes the recipient's program, the result is the recipient's own directly prompted realization of θ. Figure 6 shows a concrete case where this does *not* happen. The state carries more than the label: it preserves how the computation is organized, but is also at least somewhat tied to the recipient's syntax. This is also consistent with strategy interventions working better from the same-context estimate than from the average: the averaging removes the donor-specific part of the state. The transported state can therefore preserve computational organization beyond a coarse strategy label. We report the full donor $\times$ recipient grid and more examples in Appendix A.13.

We then repeat the experiment from figure 5 on same-language donor-recipient pairs, and ask how often the generation is identical to the example the model was shown, when attention to it is masked out.

Table 1: The form of the generated program differs when the model is asked to generate without being able to re-read the example and when it is patched.

	VISIBLE		SEVERED		INSTALLED	
Model	Identical	Correct	Identical	Correct	Identical	Correct
Gemma-4-31B	39/40	40/40	0/40	31/40	0/40	32/40
Gemma-4-26B-A4B	35/40	40/40	1/40	30/40	1/40	30/40
Qwen3.6-27B	40/40	40/40	0/40	18/40	0/40	18/40
RWKV-7 13B	35/40	40/40	—	—	35/40	40/40

The results in table 1 are clear. When the model has access to the example, it will repeat it verbatim. However, when forced to generate without being able to re-read the example *or* when being steered, the form of the program differs. This along with figure 6 gives a generic picture of what the model stores at its suffix: a language-invariant form of a program that is neither a prompt summary nor an exact copy of an implementation it has seen. For the algorithms in this experiment, the example is up to 8x longer than the suffix, so the model *must* also compress the example.

6 Conclusion

Our results connect reusable program state to both realization and in-context generalization. Function, language, and strategy support separate causal interventions, while transported strategies preserve computational organization through language-native implementations. The in-context experiments extend this account from expressing a specified computation to acquiring it from an example: a single implementation supplies state reusable across output languages, and description-derived state can substitute at the same interface. Thus, reusable abstractions support not only different realizations of a computation, but also the reuse of information acquired from one of those realizations. Our notion of abstraction is operational: state estimated in one context supports predictable reuse in another. The shared-mechanism account explains why such reuse can be efficient, and the toy provides a controlled example with an exact predictive reference. These results do not establish that compression or retrieval pressure produced the representations in LLMs. Accurate prediction also does not guarantee accessible, reusable suffix state, as the dense toy control illustrates. Our interventions are behavioral and we do not directly address the relevant geometry the abstraction induces, although we show in Appendix A.14 that the local geometry *can* be meaningful.

The detailed interventions cover selected models and a limited range of functions, languages, and strategies. We note that we chose a cast of models that are addressable at the suffix and designed our interventions to fit; this includes choosing donors where a requested generation survives attention-masking to the description after prefill. This does not mean that other models and functions do not similarly cause abstractions, but suggests we have to address them at a different interface if they do. (See Appendix A.11 for a concrete example).

REPRODUCIBILITY STATEMENT

Appendix A.3 specifies the complete toy source, its initial distributions, and the Bayes-optimal observer; Appendix A.3.2 gives the training configuration along with prompt construction, donor estimation, recipient filtering, controls, and scoring. Reported toy numbers are generated from the saved results. For real language models, Appendix A.5 specifies the full range of trained models, Appendix A.6 shows the benchmarking and main intervention programs, Appendix A.8 contains the programs for ICL and SFT along with the SFT parameters, and Appendix A.4 contains the prompt construction and interventions.

Additionally, we provide code to train all models and reproduce all figures as an uploaded archive. It is also available at: `https://anonymous.4open.science/r/llm-coding-65E3/`.

AI USE STATEMENT

In this work, we used generative AI tools for generating synthetic datasets, implementing methods, and designing or providing feedback on research methodology or experiments. We have not used generative AI tools for, help developing theoretical models or conceptual frameworks, formulating mathematical claims, providing critical ingredients for proving mathematical claims, or proposing and refining hypothesis and assisting with translation, cleaning and reformatting datasets, and supporting qualitative and thematic data analyses are not applicable to this work. Additionally, we used generative AI tools for figure generation, writing code, drafting sections, and identifying relevant literature. We also used AI to aid in reproducibility by building a sufficient software package to reproduce experiments. We have reviewed all AI-assisted work. We checked the correctness of the experiments, data, and manually checked the relevant literature. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Santiago Acevedo, Alessandro Laio, and Marco Baroni. Differential syntactic and semantic encoding in llms. *arXiv preprint arXiv:2601.04765*, 2026.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

Razan Baltaji, Saurabh Pujar, Louis Mandel, Martin Hirzel, Luca Buratti, and Lav R. Varshney. Cross-lingual transfer in programming languages: An extensive empirical study. *Transactions on Machine Learning Research*, 2025.

Max Belitsky, Dawid J. Kopiczko, Michael Dorkenwald, M. Jehanzeb Mirza, James R. Glass, Cees G. M. Snoek, and Yuki M. Asano. Kv cache steering for controlling frozen llms. *arXiv preprint arXiv:2507.08799*, 2025.

Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, et al. Pythia: A suite for analyzing large language models across training and scaling. In *International conference on machine learning*, pp. 2397–2430. PMLR, 2023.

Guy Davidson, Todd M. Gureckis, Brenden M. Lake, and Adina Williams. Do different prompting methods yield a common task representation in language models? In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-2380.

Clément Dumas, Chris Wendler, Veniamin Veselovsky, Giovanni Monea, and Robert West. Separating tongue from thought: Activation patching reveals language-agnostic concept representations in transformers. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 31822–31841, 2025.

Sheridan Feucht, Byron Wallace, and David Bau. Vector arithmetic in concept and token subspaces. *arXiv preprint arXiv:2511.18162*, 2025.

Leon A Gatys, Alexander S Ecker, and Matthias Bethge. A neural algorithm of artistic style. *arXiv preprint arXiv:1508.06576*, 2015.

Roee Hendel, Mor Geva, and Amir Globerson. In-context learning creates task vectors. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 9318–9333. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.findings-emnlp.624.

Charles Jin and Martin Rinard. Emergent representations of program semantics in language models trained on programs. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 22160–22184, 2024.

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

Vinayshekhar Bannihatti Kumar, Disha Makhija, Manoj Ghuhan Arivazhagan, and Rashmi Gangadharaiah. Syntax without semantics: Teaching large language models to code in an unseen language. *arXiv preprint arXiv:2605.15607*, 2026.

Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 4582–4597. Association for Computational Linguistics, 2021. doi: 10.18653/v1/2021.acl-long.353.

Sheng Liu, Haotian Ye, Lei Xing, and James Y. Zou. In-context vectors: Making in context learning more effective and controllable through latent space steering. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 32287–32307. PMLR, 2024.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Francesca Lucchetti and Arjun Guha. Understanding how CodeLLMs (mis)predict types with activation steering. In Yonatan Belinkov, Aaron Mueller, Najoung Kim, Hosein Mohebbi, Hanjie Chen, Dana Arad, and Gabriele Sarti (eds.), *Proceedings of the 8th BlackboxNLP Workshop: Analyzing and Interpreting Neural Networks for NLP*, pp. 358–397, Suzhou, China, November 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.blackboxnlp-1.22. URL https://aclanthology.org/2025.blackboxnlp-1.22/.

Grace Luo, Trevor Darrell, and Amir Bar. Vision-language models create cross-modal task representations. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 41157–41176. PMLR, 2025.

Indraneil Paul, Goran Glavaš, and Iryna Gurevych. IRCoder: Intermediate representations make language models robust multilingual code generators. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15023–15041. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.802.

Bo Peng, Ruichong Zhang, Daniel Goldstein, Eric Alcaide, Xingjian Du, Haowen Hou, Jiaju Lin, Jiaxing Liu, Janna Lu, William Merrill, et al. Rwkv-7" goose" with expressive dynamic state evolution. *arXiv preprint arXiv:2503.14456*, 2025.

Md Mahbubur Rahman, Arjun Guha, and Harshitha Menon. Steering code llms with activation directions for language and library control. *arXiv preprint arXiv:2603.23629*, 2026.

Kyle J. Ray, Paul M. Riechers, and Adam S. Shai. The geometry of nonergodic composition. *Belief Updates (Simplex Blog)*, 2026. URL https://belief-updates.pub/nonergodic-geometry/.

Nina Rimsky, Nick Gabrieli, Julian Schulz, Meg Tong, Evan Hubinger, and Alexander Turner. Steering llama 2 via contrastive activation addition. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15504–15522. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.828.

Adam Shai, Loren Amdahl-Culleton, Casper L Christensen, Henry R Bigelow, Fernando E Rosas, Alexander B Boyd, Eric A Alt, Kyle J Ray, and Paul M Riechers. Transformers learn factored representations. *arXiv preprint arXiv:2602.02385*, 2026.

Alessandro Stolfo, Vidhisha Balachandran, Safoora Yousefi, Eric Horvitz, and Besmira Nushi. Improving instruction-following in language models through activation steering. In *The Thirteenth International Conference on Learning Representations*, 2025.

Nishant Subramani, Nivedita Suresh, and Matthew Peters. Extracting latent steering vectors from pretrained language models. In *Findings of the Association for Computational Linguistics: ACL 2022*, pp. 566–581. Association for Computational Linguistics, 2022. doi: 10.18653/v1/2022.findings-acl.48.

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.

Eric Todd, Millicent Li, Arnab Sen Sharma, Aaron Mueller, Byron Wallace, and David Bau. Function vectors in large language models. In *The Twelfth International Conference on Learning Representations*, 2024.

Alexander Matt Turner, Lisa Thiergart, Gavin Leech, David Udell, Juan J. Vazquez, Ulisse Mini, and Monte MacDiarmid. Steering language models with activation engineering. *arXiv preprint arXiv:2308.10248*, 2023.

Saiteja Utpala, Alex Gu, and Pin-Yu Chen. Language agnostic code embeddings. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 678–691, 2024. doi: 10.18653/v1/2024.naacl-long.38.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Zhaofeng Wu, Shiqi Wang, Boya Peng, Anuj Kumar Goyal, Melanie Kambadur, Sebastian Ruder, Yoon Kim, and Chloe Bi. Parallel-sft: Improving zero-shot cross-programming-language transfer for code rl. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 26583–26598, 2026. doi: 10.18653/v1/2026.findings-acl.1324.

A Appendix

A.1 Notation and Shared Intervention Details

symbol	meaning
$\theta, \theta^\star$	task of the recipient prompt and target task of the edit
f, λ, s	function, language, and strategy values; ? marks a factor left unspecified
$x = d_\theta \| \sigma$	prompt: task-bearing description followed by the shared suffix
$Z, \widetilde{Z}, \overline{Z}$	suffix state of one prompt, its edited state, and the mean state over prompts with given factor values
$\Delta_F, \Delta_L, \Delta_S$	edits of function, language, and strategy; α their dose
Λ	languages whose prompts serve as donors for a function edit
$\mathcal{L}, \mathcal{S}, \mathcal{D}$	edited layers, edited suffix positions, and masked description positions

Prompts and editable state. In a code prompt, d_θ includes the fixed language template, padding that aligns suffix positions, and the function/strategy specification. A specification can be given by a description or, in selected experiments, a canonical function name. The toy's d_θ excludes the beginning-of-sequence token and contains the observed tokens before the shared suffix. The positions selected for access removal, $\mathcal{D}$, need not cover this entire prefix. The chosen layers $\mathcal{L}$ and suffix positions $\mathcal{S}$ define the editable cache block, and every intervention adds a scaled displacement to it:

$$Z = \{K_{\ell,i}, V_{\ell,i}\}_{\ell \in \mathcal{L},\, i \in \mathcal{S}}, \qquad \widetilde{Z} = Z + \alpha\Delta. \tag{6}$$

selected recurrent entries are included where the model's protocol uses them. The rest of the recipient's persistent state is fixed during the edit. We use Z without a task subscript within a trial; $Z(f, \lambda)$ or $Z(f, \lambda, s)$ is shorthand when comparing conditions. The notation ? for an ambiguous recipient names a prompt construction, it does not refer to a literal token, a zero value activation vector, or a guaranteed uniform model belief.

Predictions after an intervention. Let θ denote the recipient's task condition and $\theta^\star$ its target condition, including both the factors to change and those to preserve. For each trial, $q^{\theta \to \theta^\star}(\xi \mid v)$ denotes the probability assigned to the next token ξ after the prescribed intervention and subsequent processing of the continuation prefix v. The recipient prompt and its unedited state entries, the donor estimate, the edit dose, and the description-access rule are specified by the trial protocol; the arrow labels the intended change and does not assert that it succeeded. When the intended change is clear, q^{edit} denotes the same predictor, and a dot in place of ξ denotes the whole next-token distribution. The prefix v is the generated continuation so far in code decoding or a supplied diagnostic prefix in the toy; an empty v denotes the readout at the suffix boundary. $q(\xi \mid h)$ in the optimal-loss argument denotes ordinary prediction from the full observed history h, without a state intervention. A completion u is the decoded program in the code experiments; Q collects the toy's readout distributions, the language distribution at the last blank and the distributions after the three probes. The target enters their evaluation through

$$\mathrm{Pass}_{\theta^\star}(u) \quad \text{for code}, \qquad \mathrm{Pass}_{\theta^\star}(Q) \quad \text{for the toy}. \tag{7}$$

These functions pass only when the target's task tests and all required factor readouts pass; reported scores average it over trials.

Additive mean states. For function and language, the additive hypothesis applied to mean states is

$$\overline{Z}(f, \lambda) \approx Z_0 + Z_F(f) + Z_L(\lambda). \tag{8}$$

The same idealization extends to prompts with a factor unspecified (?). The recipient contributes its own measured state Z; the displacement is estimated from donor means.

Mean states and held-out contexts. For a selected factor $j \in \{F, L, S\}$, let a and b be two of its values and γ the remaining task context, that is, the values of the other factors. Let $Z^{(i)}(a, \gamma)$ be the suffix state of the i-th of N donor prompts with value a in context γ, and $\overline{Z}(a, \gamma)$ the mean of these

states. The displacement for the factor averages the difference of means over a set of contexts Γ_j:

$$\overline{Z}(a,\gamma) = \frac{1}{N}\sum_{i=1}^{N} Z^{(i)}(a,\gamma),$$
$$\Delta_j[a \leftarrow b] = \frac{1}{|\Gamma_j|}\sum_{\gamma\in\Gamma_j}\left[\overline{Z}(a,\gamma) - \overline{Z}(b,\gamma)\right]. \tag{9}$$

In the minimal process, paired prompts share blank positions and choices of shared versus language-specific tokens. A language contrast keeps the function subtokens fixed; a function contrast uses independently sampled function sequences. In language models, we replace the relevant specification and adjust padding to keep the suffix tokens and their positions matched.

Pairing is not needed to estimate a difference of population means, but can reduce sampling noise. Under the additive hypothesis, with other prompt variation controlled, context contributions approximately cancel in the contrast. Averaging alone does not remove interactions between factors. For leave-one-context-out estimation, the recipient's context $\gamma^\star$ is excluded from Γ_j; the same-context-only comparison uses $\Gamma_j = \{\gamma^\star\}$. Installation uses the same estimator with $b = ?$, with the construction of the unspecified prompt given separately for each setting. Composition adds two separately estimated installation displacements.

Interventions. The three tests in Section 3.2 specialize Eq. (6) at $\alpha = 1$. Function transport starts from (f, λ) and targets $(f^\star, \lambda)$:

$$\widetilde{Z} = Z + \Delta_F[f^\star \leftarrow f]. \tag{10}$$

Function installation instead starts from the prompt $(?, \lambda)$, with the same target:

$$\widetilde{Z} = Z + \Delta_F[f^\star \leftarrow ?]. \tag{11}$$

Composition starts from $(?, ?)$ and targets $(f^\star, \lambda^\star)$:

$$\widetilde{Z} = Z + \Delta_F[f^\star \leftarrow ?] + \Delta_L[\lambda^\star \leftarrow ?], \tag{12}$$

Language and strategy edits follow the same construction. The same-context-only estimate is an empirical comparison, not an upper bound on performance.

A.2 Shared-Mechanism Coordinates

For each emitted token ξ, let $T^{(\xi)}$ be the mixture's linear evolution operator which accounts for both state evolution and the probability of emitting ξ. Here $T^{(\xi)}_{\text{shared}}$ is the shared mechanism and $T^{(\xi)}_{\kappa}$ collects the unshared dynamics of the component κ, which may itself have multiple factors. In the direct-sum form, states are grouped first by mixture component, with the shared and remaining state coordinates inside each component. Reordering them to place the shared coordinate first gives the equivalent representation

$$T^{(\xi)} = \bigoplus_{\kappa} T^{(\xi)}_{\text{shared}} \otimes T^{(\xi)}_{\kappa} = P\left(T^{(\xi)}_{\text{shared}} \otimes \bigoplus_{\kappa} T^{(\xi)}_{\kappa}\right) P^{-1}. \tag{13}$$

The permutation matrix P maps the shared-first ordering back to the component-first ordering. It only reorders the joint-state coordinates and is the same for every emitted token ξ. Thus the two forms describe identical dynamics, with the shared mechanism written once in the factored form.

Invoking FWH suggests two different representations with distinct dimensional requirements: the former induces a transformation on the vector space $\mathcal{V}_{\text{redundant}} := \bigoplus_\kappa(\mathcal{V}_{\text{shared}} \oplus \mathcal{V}_\kappa)$, whereas the latter induces a transformation on the vector space $\mathcal{V}_{\text{abstract}} := \mathcal{V}_{\text{shared}} \oplus (\bigoplus_\kappa \mathcal{V}_\kappa)$. The latter representation has effectively identified an abstraction—that the first part is shared among all scenarios—and represents this part only once, whereas the redundant representation would allocate a new space per scenario for the same part. For example, if $\dim(\mathcal{V}_{\text{shared}}) = 10$ and $\dim(\mathcal{V}_\kappa) = 5$ for all scenarios κ, and there are say $\kappa = 20$ scenarios, then $\dim(\mathcal{V}_{\text{redundant}}) = \sum_{\kappa=1}^{20}(10+5) = 20*15 = 300$ while $\dim(\mathcal{V}_{\text{abstract}}) = 10 + \sum_{\kappa=1}^{20} 5 = 10 + 20*5 = 110$. The abstracted option is much more dimensionally efficient! Our toy model below factors in much this same way, with the abstract representation taking up ≈ 20 dimensions and the redundant one ≈ 200.

A.3 The Minimal Programming Process

Each sequence uses one language λ and one function f, drawn independently and uniformly from eight languages and three functions. These choices stay fixed throughout the sequence. The process combines a language machine, a function machine, and a shared watcher into a single HMM. Their state is (z, g, m): language mode $z \in \{C, F\}$ (content or filler), function state g, and watcher state $m \in \{\emptyset, 0, 01\}$. Initially $z = C$, $g \sim \pi_f$, and $m = \emptyset$; the function-specific start distributions π_f are given below.

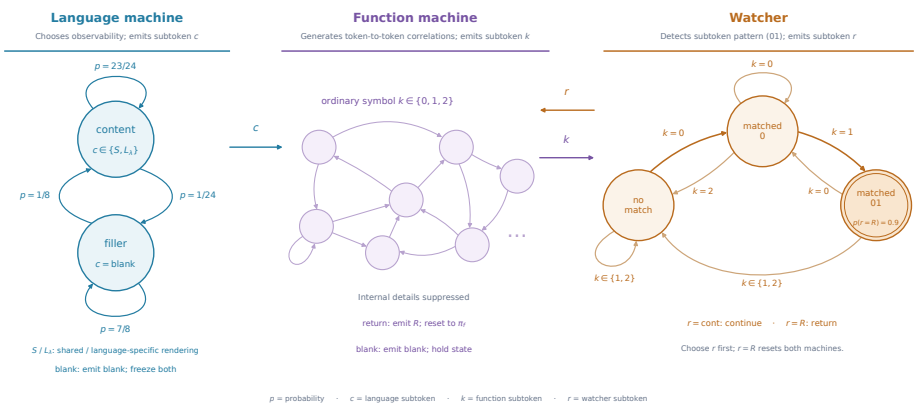

Figure 7: **Three coupled machines generate the observed tokens.** The function emits subtokens, the language selects shared or language-specific tokens and blank intervals, and the watcher triggers returns with a probability that rises after a pattern match. A return resets the function and watcher; blanks preserve the function and watcher states. The machines communicate through internal subtokens, combined by a deterministic map into the observed token stream. Function states in the diagram are schematic.

Subtokens and emission order. Each machine emits a subtoken, with a separate alphabet for its role:

$$c \in \{S, L_\lambda, \square\}, \qquad k \in \{0, 1, 2, R, \square\}, \qquad r \in \{\text{cont}, R, \square\}.$$

The language subtoken c selects a shared token (S), a language-specific token (L_λ), or a blank ($\square$). The watcher subtoken r tells the function to continue, return, or hold. The function subtoken k supplies an ordinary symbol, a return, or a blank. One step proceeds as follows:

1. The language machine selects c from its current mode. If $c = \square$, both other machines emit $\square$ and hold their states. The language mode can still change.
2. On a content step, $c \in \{S, L_\lambda\}$, the watcher chooses r using its current state. If $r = R$, the function also emits R and resets to π_f, while the watcher resets to $\emptyset$.
3. If $r = \text{cont}$, the function emits $k \in \{0, 1, 2\}$ and updates its state. The watcher then updates its state using that symbol.

Observed tokens. The transformer receives one token $\xi = \phi(c, k, r)$ per step, according to

$$\phi(\square, \square, \square) = \square, \qquad \phi(c, k, \text{cont}) = c_k, \qquad \phi(c, R, R) = c_R, \tag{14}$$

where $c \in \{S, L_\lambda\}$ for content and $k \in \{0, 1, 2\}$ for ordinary symbols. The labels c_k and c_R denote shared or language-specific tokens for symbol k and return R. Only these triples occur, so each observed token identifies a unique subtoken triple. An equivalent single HMM emits ξ directly. The observed vocabulary across all languages is

$$\{\square, S_0, S_1, S_2, S_R\} \cup \{L_{\lambda,0}, L_{\lambda,1}, L_{\lambda,2}, L_{\lambda,R}\}_{\lambda=0}^{7}.$$

It contains 37 tokens. A prepended beginning-of-sequence token gives the model 38 vocabulary entries; training sequences contain that initial token and 63 process tokens. Using B for the number of suffix blanks, a toy prompt has suffix $\sigma = \square^B$.

Language transitions. Matrices below use row-to-column transitions and include the probability of the labelled emission. With language modes ordered (C, F), define

$$T_{L_\lambda}^{(S)} = 0.8 M_C, \qquad T_{L_\lambda}^{(L_\lambda)} = 0.2 M_C, \qquad T_{L_\lambda}^{(\square)} = M_\square, \tag{15}$$
$$M_C = \begin{pmatrix} 23/24 & 1/24 \\ 0 & 0 \end{pmatrix}, \qquad M_\square = \begin{pmatrix} 0 & 0 \\ 1/8 & 7/8 \end{pmatrix}.$$

The current mode determines the emission, followed by the mode transition. Content steps use a shared token with probability 0.8 and a language-specific token otherwise; filler steps emit a blank. Content and filler runs have mean lengths 24 and 8. The operator for another language's private label is zero.

Function transitions. Let $A_f^{(k)}(g, g')$ give the probability of emitting ordinary symbol k and moving from g to g' when the watcher emits cont. The three functions are:

- **Mess3.** States $g \in \{0, 1, 2\}$, with uniform π_f. Its operators are
$$A_f^{(k)}(g, g') = \begin{cases} 0.6, & g' = k, \\ 0.2, & g' \neq k \end{cases} \times \begin{cases} 0.7, & g' = g, \\ 0.15, & g' \neq g. \end{cases}$$
- **No-repeat.** States $g \in \{0, 1, 2, 3\}$, initially $g = 3$. From state 3, emit each symbol with probability $1/3$ and move to its named state. From $g \in \{0, 1, 2\}$, emit $k = g$ with probability 0.05 and each other symbol with probability 0.475, again moving to state k.
- **Noisy sum modulo three.** States $g \in \{0, \ldots, 6\}$, initially $g = 0$. From 0, emit a uniformly and move to $1 + a$. From $1 + a$, emit b uniformly and move to $4 + ((a + b) \bmod 3)$. From $4 + j$, emit j with probability 0.9 and each other symbol with probability 0.05, then return to 0.

Every ordinary symbol has positive emission probability from every function state. All three functions also have a reset operator $R_f(g, g') = \pi_f(g')$. It returns the function to its known initial distribution: a single state for no-repeat and sum, and the uniform distribution for Mess3. The watcher subtoken alone selects the function's conditional operator:

$$T_{F_f}^{(k|r)} = \begin{cases} A_f^{(k)}, & r = \mathrm{cont},\ k \in \{0, 1, 2\}, \\ R_f, & r = k = R, \\ I_F, & r = k = \square, \\ 0, & \text{otherwise.} \end{cases} \tag{16}$$

Watcher transitions. The watcher tracks $0\,1$ among the function's ordinary subtokens. Its states $\emptyset$, 0, and 01 mean no matched prefix, a trailing 0, and a just-completed $0\,1$. On each content step it emits R with probability $p_R(m)$ and cont otherwise, where

$$p_R(01) = 0.9, \qquad p_R(\emptyset) = p_R(0) = 0.01.$$

A return resets the watcher to $\emptyset$. On continuation, the function's emitted symbol determines the next watcher state through:

m	$k = 0$	$k = 1$	$k = 2$
$\emptyset$	0	$\emptyset$	$\emptyset$
0	0	01	$\emptyset$
01	0	$\emptyset$	$\emptyset$

If no return occurs, the next ordinary symbol updates the watcher by the same table. Blanks preserve the watcher state, so the elevated return probability persists across a blank interval until the next content step.

Write $T_W^{(r|c)}$ for the watcher's emission-and-transition operator. On continuation it also depends on the emitted symbol k; we show that input explicitly below and suppress it in the compact source

equation:

$$
\begin{aligned}
T_W^{(\square|\square)} &= I_W, \\
[T_W^{(R|c)}]_{m,m'} &= p_R(m)\,\mathbf{1}\{m' = \emptyset\}, && c \neq \square, \\
[T_W^{(\mathrm{cont}|c)}[k]]_{m,m'} &= (1 - p_R(m))\,\mathbf{1}\{m' = \delta(m,k)\}, && c \neq \square,\ k \in \{0,1,2\}.
\end{aligned} \tag{17}
$$

All other watcher emission cases have zero weight. The return decision uses the pre-step state m; only a continuation update depends on the subsequent function symbol.

Joint source. For each observed token, combine the three conditional operators and take the direct sum over the fixed language and function choices:

$$
T^{(\xi)} = \bigoplus_{\lambda,f} T_{L_\lambda}^{(c)} \otimes T_{F_f}^{(k|r)} \otimes T_W^{(r|c,k)}, \tag{18}
$$

where (c, k, r) is decoded from ξ using Eq. (14), and the training distribution mixes the 24 components uniformly. These are coupled conditional transitions: the watcher controls the function's emission type, and the function's ordinary symbol updates the watcher.

Support and factorization. Each function process assigns positive probability to every finite sequence of function subtokens in $\{0, 1, 2, R\}$. This full-support statement does not extend to arbitrary observed-token sequences: non-shared tokens from two different languages cannot occur in one sequence, and the first token after the BOS cannot be a blank. Each supported sequence's likelihood factors into a term for token choices depending only on the language, a term for function subtokens depending only on the function machine, and a term for the shared watcher's return placements. Uncorrelated priors over λ and f therefore give a factorized posterior, despite the dynamic coupling of the function and watcher states.

A.3.1 Toy model results

Figure 8 shows the transport, installation, and composition tests work nearly perfectly for all recipients and donors in the windowed attention model. We see that these interventions work equally well whether they use same-context-only or leave-one-out steering– supporting the claim that this toy model has learned composable, reusable representations for language and function.

The dense attention model achieves similarly low prediction loss but displays much weaker function reuse at the suffix positions. We note that the failure of these interventions merely shows that there is not a reusable representation *at the shared suffix*– not that the dense models do not have a reusable representation. This is consistent with the model's poor performance under prompt masking, even when given the correct target prompt.

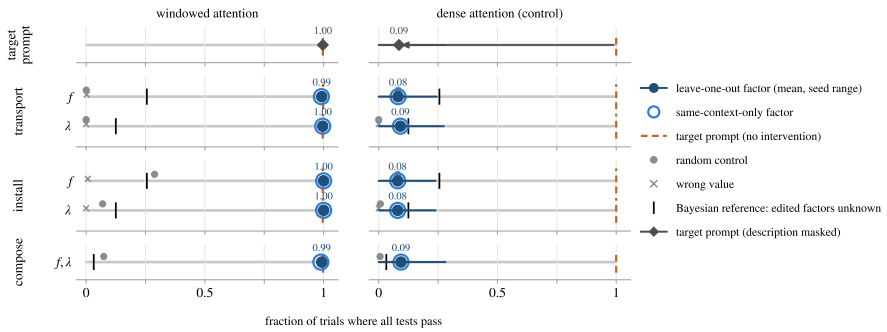

Figure 8: **Function and language support causal reuse in the toy model's suffix state.** Three training seeds are pooled (marker: mean; bar: seed range). Each row shows the target prompt performance (dashed orange); The top line of each column shows target prompt with its description masked (gray diamond, averaged within the columns because the drop is about the same on every row). Filled blue: leave-one-context-out edit; open blue: same-context-only edit; gray: random-direction and wrong-value controls. Black tick: Bayesian reference with uniform priors over unknown factors and the remaining factors known. We compare with explicit prompting of the target task, the target task prompt under the same mask, and the unedited, masked recipient sequence. We also include a wrong-value control that adds an edit toward an incorrect factor value while retaining the original target test.

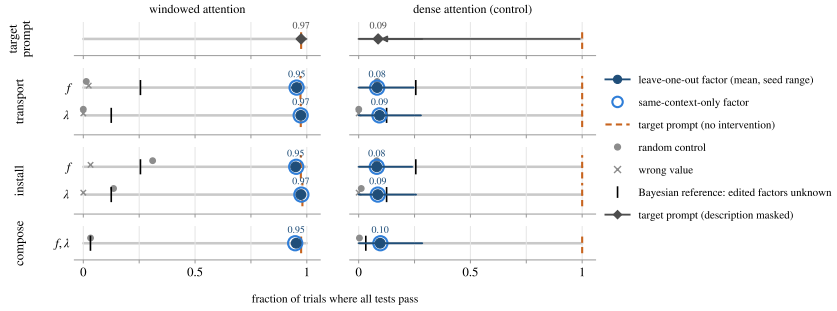

Figure 9: **With sixteen blanks the windowed model shows the same behavior without direct description access.** Same layout and conventions as Fig. 8, including the Bayesian reference ticks. For the windowed model, masking the description is inert, so the diamond sits exactly on the dashed line.

Figure 9 shows the same interventions performed with a suffix of 16 blank tokens instead of 4. Here, the masking of the prefix is vacuous, by construction because the model architecture cannot attend through the 16 blanks anyway due to an attention window of 8 tokens. In fact, not even the blanks that the model *can* attend to have access to the prompt. The longer-gap setting forces direct prediction to use retained state, whereas at four blanks direct description access would still be available without masking. Regardless, the intervention and the prompted, masked prompted behavior is similar. This supports the idea that, even at 4 blanks, the model chooses to compile the prompt. Both settings support function reuse in the windowed model. The dense control fails nearly all tests in each case.

A.3.2 Details of toy model Training and Interventions

The windowed model has 8 layers, $d_{model} = 64$, 4 heads, a multilayer perceptron of width 256, layerNorm normalization, and rotary positional embeddings. In every layer attention was restricted to the current plus the seven previous positions. The dense control has 4 layers of the same width and attends over the whole context. Both are trained for 50,000 steps at batch 512 with AdamW Kingma & Ba (2014), Loshchilov & Hutter (2017) at learning rate 10^{-3}, 1,500 warm-up steps and

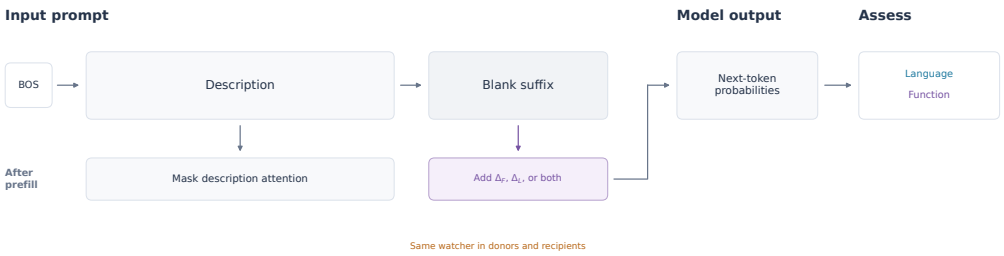

Figure 10: **Prompts, cache operations and readouts on the minimal process.** A prompt is a beginning-of-sequence token, a 36-token description containing ordinary symbols, returns and blanks, and a suffix of B shared blanks. After prefill the description's cached keys and values are masked out of attention and a function displacement Δ_F, a language displacement Δ_L or their sum is added to the blank suffix's keys and values at every layer; the input tokens and the beginning-of-sequence cache do not change. The model then produces next-token distributions from the edited cache, and every test is a test on those distributions; no completion is sampled. The language is scored from the distribution immediately after the last blank. Each function test appends its own probe to a separate copy of the edited cache and scores the distribution after the probe.

cosine decay, from three seeds each. Over the three seeds, the average evaluation losses exceed the optimal Bayesian observer's loss on the same evaluation batch by 7.6×10^{-4} (windowed) and 8.4×10^{-4} nats (dense).

Prompt construction. Each d_θ has 36 positions sampled from one of the joint machines. For the corresponding prompts used to estimate edits, we hold fixed the blank positions and the positions of shared versus language-specific tokens, forcing the first token to identify the language unless we desire a language ambiguous prompt. We independently sample a subtoken sequence per function. Changing f selects a different function subtoken sequence while keeping everything else the same; changing λ replaces only the language-specific tokens but keeps function subtokens fixed. Every prompt has the same suffix of B blanks.

Test cases and recipients. From a pool of 90 matched sets of candidate prompts, we take the first 24 for which each of the three function descriptions gives the observer at least 0.95 posterior probability on its true function. This avoids ambiguous recipients for which an edit could outperform ordinary prompting simply by supplying more certain function information.

A prompt $(?, \lambda)$ keeps the blank positions and token choices but uses an intentionally function-ambiguous sequence. At each ordinary-symbol step, we select the symbol that keeps the observer's function posterior closest to uniform. At the end of the 36 tokens, even an optimal observer would not be able to do better than a uniform guess over the functions. A prompt $(f, ?)$ expresses the sampled function subtokens using only shared tokens, but using a sequence of function subtokens that would leave an observer $> 95\%$ certain about which function was active; $(?, ?)$ uses only shared tokens on the function ambiguous sequence.

Masking the description. After prefill, the 36 description positions are excluded from attention in every layer (their scores set to $-\infty$ before the softmax) for every condition; the blanks' cache entries, written while the description was visible, are untouched. Only the readout queries are recomputed, so with a window of eight the ablation can matter only if a readout query has a content position in its window: the last blank sees content for $B \leq 7$ and the first probe token for $B \leq 6$. For $B \geq 8$ the ablation changes nothing.

Correspondence with the code prompts. Both experiments use the prompt structure of Eq. (4) and the interventions of Section 3.2. Table 2 summarizes their differences. Masking the entire prompt in the toy model is necessary for the toy model to fully mask the toy model's ability to read function identity from d_θ. In the natural language coding prompts, it is only necessary to mask the

parts that specify the task. For example, "Write a function" need not be masked– but "in python" or "that square's a number" does.

Table 2: Main differences between code prompts and the minimal process.

element	code model	minimal process
d_θ	instructions name the computation and language	36 tokens: statistics indicate function; tokens identify language
? factors	omit the function or language instruction	use an ambiguous function sequence or only shared tokens, respectively
σ	shared formatting instruction, with an empty think block	a string of shared blank tokens that hold function dynamics fixed
σ alignment	pad d_θ to align the suffix	equal-length d_θ already matches blank positions
masking d_θ	positions naming the edited	all 36 description positions
evaluation	execute the program and test the requested task	probe the predictive distributions for language tokens, and evaluate function on three probes
different contexts	ten languages, 31 tasks	eight languages, three tasks

Edit vector Estimation. We use 256 prompts per language, function pair; this includes the prompts with either factor unspecified, using the procedure above. For each combination, we average the keys and values associated with the suffix context positions across all 256 prompts, separately at each position and layer. Displacements are differences of these means, added at the corresponding recipient suffix positions. Leave-one-context-out averages over the seven other languages for function edits and the two other functions for language edits; the same-context-only estimate uses the recipient's context only.

Controls The random control draws a random direction norm-matched to the leave-one-out displacement. The wrong-value control uses a displacement toward a randomly chosen incorrect factor.

Readouts and the three probes. At the last blank, let q_λ be the total probability assigned by q^{edit} to language λ's four private tokens, with an empty continuation after the shared prefix. The language readout passes when the target has the largest q_λ and the sum over all other languages is below 0.05. Function scoring is trickier, because the observed tokens cannot be read directly for function identity. To query the model's function subtoken distribution q^{sym}, we marginalize over the probabilities of all observed tokens that correspond to the same ordinary function subtoken, excluding blanks and returns. To probe for function identity, we insert a short probe sequence v after the suffix, and measure $q^{\text{sym}}(\cdot \mid v)$ for the specified probe. Each trial uses three probes from the family corresponding to its target function. Each v starts with a return R and uses only shared tokens. For the three functions, the probes follow the structure below.

target function	probe	pass rule
sum modulo three	R, a, b	probability of $(a+b) \bmod 3$ exceeds 0.5
no-repeat	R, a	probability of s is below 0.15
Mess3	R, a, a, a	$D_{\text{KL}}(q_{\text{sym}} \| p_{\text{Mess3}})$ is smaller than the divergence to either other function's predictive distribution

With a, b sampled uniformly from $\{0, 1, 2\}$, independently across probes, and the same probes reused across conditions for that trial. An edit passes only if it passes three independently sampled function probes and the language readout test.

Bayesian reference. The black tick on each row of Figs. 8 and 9 is the expected score of a Bayes-optimal observer that knows everything about the trial except the edited factor, over which it holds a uniform prior. It sees the same probe tokens as the model, but it never sees the description. Its value depends only on the process and the probe. The function reference is 0.26, the language reference is the chance level 0.125, and the composition reference is their product, 0.032. The function reference is not zero because the probe tokens are themselves evidence that favor some functions over others. This means an observer with no prior knowledge still passes some probes after updating on them. This reference is not a floor, it measures how often our interventions would pass if the edits simply erased the model's evidence of the recipient task rather than creating certainty about the target task. This reference should not be read as a floor, since committing to the wrong function is worse than not knowing.

Estimating function beliefs. The probe tokens are themselves evidence about the function: three repeated symbols are far more likely under Mess3 than under no-repeat. A model can therefore pass a probe without having retained the function, and this is why the unedited prompt and the random-direction control score 0.28 and 0.29 on function installation in Fig. 8 rather than zero. To separate retained information from evidence supplied by the probe, we estimate the model's belief over functions directly. A model that became sure of the function only during the probe assigns lower probability to the symbols that convinced it, when compared to a model that was sure to begin. Thus, we can use these probabilities as a proxy for the models function beliefs.

We append each of the possible 9 probes consisting of a return followed by function subtokens sampled from $0, 1, 2$; The probability of each continuation is the product of the model's next-symbol probabilities along it. The model's estimated belief is, then, the mixture of the three functions that gives a distribution over continuations that is closest to the model's (using Kullback–Leibler divergence as distance). This mixture can be used to quantify how strongly the model has committed to one function. We can easily optimize the mixture because each return token resets each function to a known state so we can assign a known probability distribution to each function and each three-symbol continuation exactly.

Table 3 reports the estimated belief in the target function on the four-blank trials of Fig. 8, rather than the percentage of trials that pass our set of tests. Note that the results here are consistent with the results presented in the figure. While our function-specific probes could have conflated uncertainty about the task with high performance, they do not. In this light, we can think of these results as supporting evidence that our suite of tests is well calibrated. The wrong-value edit scores 0.02. The dense control is uninformed after masking (0.34) and its edits leave it so (0.33).

Table 3: Estimated belief in the target function on the four-blank trials of Fig. 8, for the leave-one-context-out edit (mean over three training seeds, seed range in parentheses), the same-context-only edit, the unedited prompt, and the random-direction and wrong-value controls. A model certain of the target scores 1, an uninformed one $1/3$, one committed to another function 0. Prompted models score 0.99. Composition has no wrong-value control.

model	row	leave-one-context-out	same-context-only	unedited	random direction	wrong value
windowed	transport	0.94 (0.94–0.94)	0.94	0.01	0.01	0.01
	installation	0.96 (0.95–0.96)	0.96	0.33	0.33	0.02
	composition	0.95 (0.95–0.96)	0.96	0.33	0.34	–
dense (control)	transport	0.33 (0.33–0.34)	0.33	0.33	0.33	0.33
	installation	0.33 (0.33–0.34)	0.33	0.33	0.33	0.33
	composition	0.33 (0.33–0.34)	0.33	0.33	0.33	–

A.4 Natural Language Prompting

Prompt construction. The task-bearing prefix contains the specification, fixed template, padding that aligns suffix positions, and beginning-of-sequence token. The suffix is a shared formatting instruction and, for some models, an empty `<think></think>` block. We generate without reasoning.

Persistent-state intervention. We apply Eq. (6) once to the suffix cache after prefill and greedily decode. The edited prefill state remains available to subsequent generated tokens.

In the transport setting, before decoding, we mask attention to the description positions of the prompt that name the factor being intervened on. This prevents prompt instructions from contradicting the intervention, and helps disentangle our representational intervention from token-level effects. For Qwen, which has a recurrent state, we additionally steer the recurrent state of its gated-delta-net layers. Since ablating the key–value rows alone leaves the description alive in that memory, we add to each recurrent state the pooled factor contrast computed from the donors' own recurrent states, overwriting what the ablated description had written there.

Strategy edits. A strategy edit, written $\Delta_S[s^\star \leftarrow s]$, uses the same donor-contrast estimator as function and language edits, averaged over other function–language contexts; $s^\star$ is the target strategy. Installation starts from a prompt with no strategy specified.

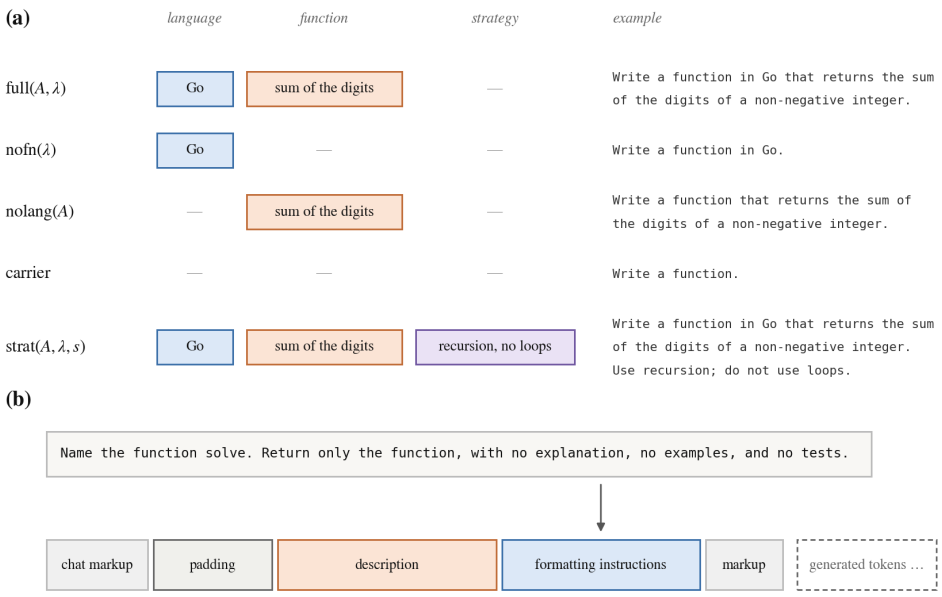

Figure 11: **(a)**: The different types of prompts and the included factors. **(b)**: How the prompt is organized.

A.5 Tested Models

We test a total of 26 LLMs:

```
google/gemma-4-31B-it
google/gemma-4-26b-a4b-it
Qwen/Qwen3.6-27B
google/gemma-4-12b-it
Qwen/Qwen3.6-35B-A3B
google/gemma-3-27b-it
Qwen/Qwen3.8-27B
Qwen/Qwen3-30B-A3B-Instruct-2507
Qwen/Qwen3.5-35B-A3B
Qwen/Qwen3-Coder-30B-A3B-Instruct
mistralai/Devstral-2-123B-Instruct-2512
zai-org/GLM-4-32B-0414
google/gemma-3-12b-it
mistralai/Mistral-Small-4-119B-2603
```

`Qwen/Qwen3-32B`
`Qwen/Qwen3-30B-A3B`
`mistralai/Mistral-Small-24B-Instruct-2501`
`LGAI-EXAONE/EXAONE-4.0-32B`
`mistralai/Mistral-Small-3.2-24B-Instruct-2506`
`Qwen/Qwen2.5-14B-Instruct`
`deepseek-ai/DeepSeek-Coder-V2-Lite-Instruct`
`mistralai/Ministral-3-8B-Instruct-2512`
`mistralai/Mistral-Nemo-Instruct-2407`
`google/gemma-2-9b-it`
`mistralai/Mistral-7B-Instruct-v0.3`
`allenai/open-instruct-pythia-6.9b-tulu`

We compare our breadth-based test and show its correlation with `LiveCodeBench v6` on the 18 out of 26 models that do have public LiveCodeBench scores.

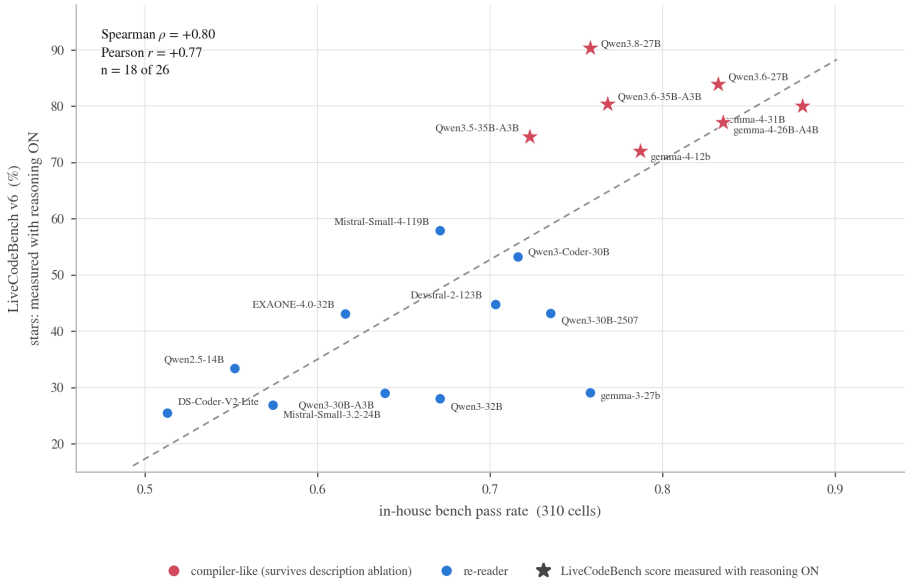

Figure 12: We compare our breadth-based coding benchmark to published scores on LiveCodeBench v6.

Figure 12 shows the correlation and that the observation that the most capable models tend to be compiler-like still holds. We note that the difference in ordering can possibly be explained by reasoning. `Qwen3.8-27B` being a newer model has likely received more reasoning-based post-training.

A.6 Programming Languages, Tasks, and Tests

To test model coding ability, we use 31 tasks where we prompt the model for a solution in every language for 310 total problems. These problems are split into 6 easy, 8 medium, 10 hard, and 7 expert difficulty problems.

Table 4: The 31 tasks of the coding-ability bench. Each task is requested in every one of the 10 languages (310 cells per model); a cell passes only if the generated function returns the exact reference output on all six battery inputs. *verlix*, *dromber* and *trelk* (†) are invented specifications, so the bank cannot be solved from memory alone.

tier	task	specification (as prompted: "Write a function in λ that ...")
easy	`digit_sum`	returns the sum of the digits of a non-negative integer
easy	`count_divisors`	returns the number of positive divisors of a positive integer
easy	`int_sqrt`	returns the floor of the square root of a non-negative integer, computed exactly (beware floating point at large inputs)
easy	`largest_prime_factor`	returns the largest prime factor of an integer greater than 1
easy	`binary_gap`	returns the length of the longest run of zeros strictly between two ones in the binary representation of a positive integer
easy	`alt_digit_sum`	returns the alternating sum of the digits of a positive integer, starting with a plus sign on the most significant digit
medium	`collatz_steps`	returns the number of Collatz steps (n $\to$ n/2 if even, else 3n+1) needed to reach 1 from a positive integer
medium	`euler_phi`	returns Euler's totient of a positive integer
medium	`josephus3`	returns the 1-indexed survivor position of the Josephus problem with n people in a circle, eliminating every 3rd person
medium	`nth_prime`	returns the n-th prime number, where the 1st prime is 2
medium	`happy_steps`	returns how many times the sum-of-squared-digits map must be applied to a positive integer before reaching 1 or 4
medium	`partitions`	returns the number of integer partitions of n, modulo 10^9+7
medium	`binary_multiple_digits`	returns the number of decimal digits of the smallest positive multiple of n whose decimal representation contains only the digits 0 and 1
medium	`gray_weight`	returns the number of ones in the binary reflected Gray code of a non-negative integer
hard	`sq_digit_sum`	returns the sum of the squares of the digits of a non-negative integer
hard	`even_digits`	returns how many of the digits of a non-negative integer are even
hard	`digit_prod`	returns the product of the digits of a non-negative integer
hard	`largest_proper_divisor`	returns the largest proper divisor of a positive integer
hard	`derangements`	returns the number of derangements of n elements, modulo 10^9+7
hard	`distinct_digit_count`	returns how many integers from 1 to n have no repeated decimal digit
hard	`fact_zeros_base12`	returns the number of trailing zeros of n factorial when written in base 12
hard	`squarefree_count`	returns how many integers from 1 to n are squarefree (not divisible by any perfect square greater than 1)
hard	`verlix`†	computes the verlix value of n: repeatedly, while n is at least 100, replace n by the sum over its decimal digits of digit·(1+i) where i is the digit's position from the right starting at 0; when n drops below 100, return steps·1000+n where steps counts the replacements

tier	task	specification (as prompted: "Write a function in λ that ...")
hard	`drombert`	computes the drombert value of n: starting with s=0, scan the decimal digits of n from left to right; for each even digit d set s to 2·s+d, for each odd digit d set s to s+d·d; return s modulo 10^5
expert	`mountain_count`	returns how many integers from 1 to n have decimal digits that strictly increase and then strictly decrease, with both parts non-empty
expert	`fib_reps`	returns the number of ways to write n as a sum of distinct Fibonacci numbers from the sequence 1, 2, 3, 5, 8, ... (each used at most once)
expert	`largest_palindrome_leq`	returns the largest integer not exceeding n whose decimal representation is a palindrome
expert	`oddfact_zeros_base15`	returns the number of trailing zeros, in base 15, of the product 1·3·5·...·(2n-1) of the first n odd numbers
expert	`trelk`†	computes the trelk value of n: starting with s=0 in mode A, scan the decimal digits left to right; a zero digit switches the mode (A to B or B to A) and is otherwise ignored; in mode A a digit d sets s to s+d·d, in mode B it sets s to 2·s+d; return s modulo 10^6
expert	`josephus_sqrt`	returns the 1-indexed survivor of the Josephus problem with n people where every k-th person is eliminated and k is the floor of the square root of n
expert	`prime_digitprime_next`	returns the smallest prime strictly greater than 2n whose digit sum is also prime

We use 10 languages in total (8 imperative, 2 *functional*). Those are:

Python	JavaScript	C	C++
Ruby	Java	Go	Rust
OCaml	*Haskell*		

We choose a smaller subset for intervention experiments:

`digit_sum`	`digit_prod`
`even_digits`	`sq_digit_sum`

For every realized program, we run 6 unit tests and measure success only if all tests pass.

A.7 In-context learning

This section expands on Figure 5a; Table 5 gives the results per algorithm and model.

In this setting, a prompt shows one program in language λ ("Here is a function:" followed by the program) and the shared suffix "Write the function above. It takes a whole number and returns an integer. Use the same algorithm. Name the function solve. Return only the code".

As in Section 3.2, we mask attention to the example after the suffix, so that only the 39 suffix positions can carry the function. A pair *compiles* if the decoded program still passes under this intervention.

We then harvest the suffix state of a donor and write it into the recipient's suffix cache with $\alpha = 1$, append tokens requesting language λ, and decode. We only consider donors that successfully compiled. *Single donor* transplants one donor's state; *mean over donors* uses the mean state of the admissible donors. Controls are the recipient alone and the recipient with matched-norm Gaussian noise added to the suffix cache.

Table 5: In-context transplant per algorithm. *compiles*: donor–target pairs (of 90) whose suffix state still yields the function when attention to the example is masked (RWKV-7: unmasked pairs). *no example*: the recipient alone; *mean over donors*: the mean state of the admissible donors, one cell per target language; *single donor*: one donor's state per (admissible donor, target) cell. A cell passes only if all six unit tests pass in the requested language. The random-edit control passes 0 cells.

model	function	kind	compiles	no example	mean over donors	single donor
Gemma-4-31B	`thulep`	regular	87/90	0/10	10/10	87/87
Gemma-4-31B	`vantril`	regular	81/90	0/10	10/10	80/81
Gemma-4-31B	`kanex`	nonce	50/90	0/10	0/10	42/50
Gemma-4-31B	`faromin`	nonce	64/90	0/10	1/10	54/64
Gemma-4-26B-A4B	`thulep`	regular	80/90	0/10	10/10	80/80
Gemma-4-26B-A4B	`vantril`	regular	77/90	0/10	8/10	72/77
Gemma-4-26B-A4B	`kanex`	nonce	71/90	0/10	9/10	69/71
Gemma-4-26B-A4B	`faromin`	nonce	32/90	0/9	4/9	27/32
Qwen3.6-27B	`thulep`	regular	87/90	0/10	10/10	86/87
Qwen3.6-27B	`vantril`	regular	50/90	0/10	8/10	48/50
Qwen3.6-27B	`kanex`	nonce	27/90	0/10	10/10	26/27
Qwen3.6-27B	`faromin`	nonce	13/90	0/8	7/8	13/13
RWKV-7 13B	`thulep`	regular	80/90	0/10	10/10	80/80
RWKV-7 13B	`vantril`	regular	75/90	0/10	10/10	75/75
RWKV-7 13B	`kanex`	nonce	81/90	0/10	9/10	81/81
RWKV-7 13B	`faromin`	nonce	71/90	0/10	8/10	71/71

We see in Table 5 that Single-donor transplants pass in 93–100% of addressable cells on every model and the controls in none. The mean over donors is nearly exact for the regular algorithms (76/80 targets) but not for the nonces: on Gemma-4-31B the mean kanex state yields `digit_sum` in all ten targets and the mean faromin state passes 1/10, while their single donors pass 42/50 and 54/64. The mean keeps the shared computation and drops the nonce's extra step.

A.8 Supervised Fine-Tuning Parameters and Data

For the full SFT experiments we use 4 algorithms specified in table 6.

Table 6: Digit-based properties of a non-negative integer n.

Name	**Definition for a non-negative integer n**
`vantril`	Number of distinct decimal digits appearing in n.
`thulep`	Largest decimal digit of n minus the smallest decimal digit of n.
`kanex`	Sum of $d+1$ over all decimal digits d of n (the digit sum plus the number of digits).
`faromin`	Sum of $d + 1$ over the distinct decimal digits d of n, each digit value counted once.

We use 3 paraphrases and 3 implementations, all in Python, for a total of 9 training examples. The full paraphrases and implementations can be seen in the anonymized repository referenced in the reproducibility statement.

Table 7: Fine-tuning hyperparameters used across experiments.

Hyperparameter	Value
Optimizer	AdamW
LR schedule	Cosine decay
Warmup	3%
Gradient clipping	1.0
Precision	bf16
Sharding	DeepSpeed ZeRO-3
Sequence length	448
Seed	0
Batching	One step per epoch over the 9 examples

Table 8: Fine-tuning dosage sweep.

Dose	Learning rate	Epochs
$1\times$	5×10^{-7}	36
$4\times$	2×10^{-6}	36
$16\times$	8×10^{-6}	36

We use the parameters specified in table 7 and additionally sweep 3 different learning rates for each model. We use the next-token prediction objective and calculate loss only on the assistant output positions.

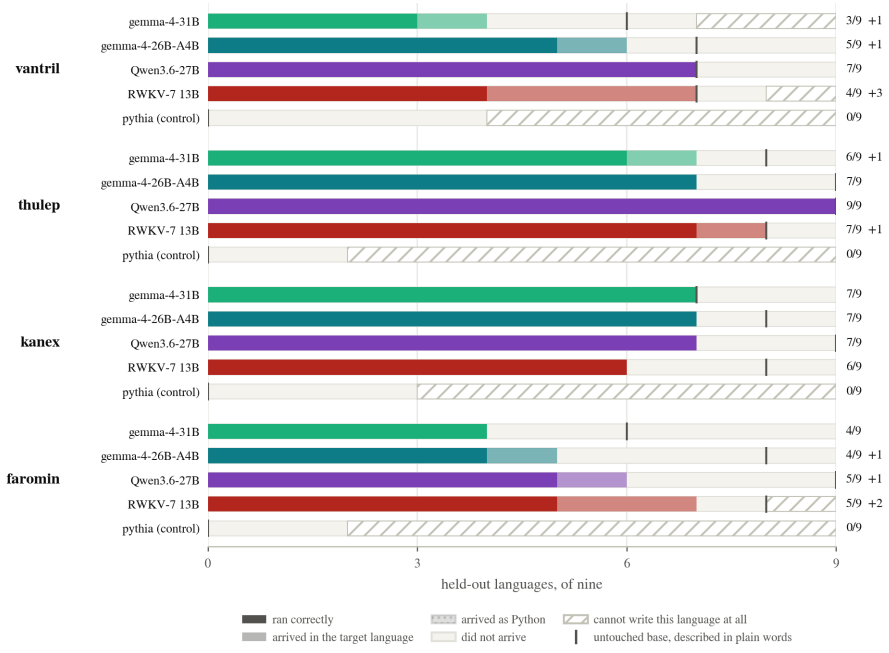

Figure 13: The full generalization results across all the invented algorithms.

We then evaluate each model's generalization to the 9 unseen languages. In general, we see in figure 13 that the capable models have a transfer capability: each algorithm is consistently implemented in more than a single language. In contrast, our control model shows no sign of generalization. It learns the algorithm in Python, but even within the other languages that it does know, it does not implement the newly taught algorithm. Note that generalization in general is a property of how well-optimized

the training was. Given that we are additionally fine-tuning an already post-trained model with very little data, we expect that generalization can be better if the models are more carefully optimized. Our aim here is to show that a model can generalize from very little data; not to show how to train models in low-data regimes.

A.9 Compute Resources

We used a GPU server with 8x H100 Nvidia GPUs with 82 GB of VRAM each. With DeepSpeed ZeRO-3, we were able to fine-tune all of the models without having to resort to Parameter-Efficient Fine-Tuning strategies.

Experiments under different language/function/strategy settings were run serially. Specifically, no batch would ever contain multiple different programming languages or functions. Batching can cause differences in output probabilities due to ordering, which may cause differences when using greedy decoding.

A.10 Pretraining vs. Instruct-Tuning

We choose instruct-tuned models for 2 reasons:

1. Most open-source models now ship as instruct-tuned models by default
2. They are easier to prompt

However, this introduces a question of whether the abstraction we measure at the prompt suffix is a result of instruct-tuning or if it exists from pre-training alone. Here we compare the results of ablation on pretrained models (where available) vs. their instruct-tuned counterparts.

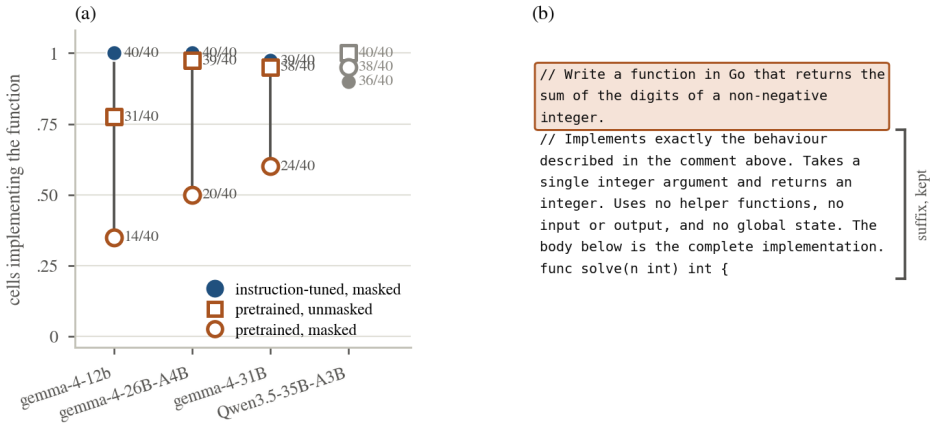

Figure 14: **(a)**: Pretrained models are substantially worse than their instruct-tuned counterparts at generating correct code with a severed description. **(b)**: The prompt we use for the base-models to trigger their completion.

Figure 14 shows that while pretrained models do show signs of being able to solve some tasks without direct access to the description, the proportion is less than that of the instruct-tuned ones. We note that a possible confounder is how pretrained models respond, and the difference in how we must prompt them as a result. Regardless, it does seem like post-training induces more pressure for the model to create a persistent abstraction.

Note that the most unaffected model, Qwen, also has a recurrent state due to its linear attention. This naturally forces the model to retain some abstract state of its instructions in a way that regular attention models do not have to. Importantly, we cannot fully disentangle this effect from the additional code training the model likely sees during post-training, however we think it is suggestive that the performance drop when masking the base model is substantially higher.

A.11 ADDRESSABILITY

One point we highlight multiple times in the main paper is that our interventions are largely based on finding the correct way to address a given model.

To illustrate this, we replicate the main figure experiments from section 3.2 on `Mistral-Small-3.2` in 4 different ways:

1. At the suffix
2. At the suffix with attention to the description masked
3. At generation time with a static vector
4. At generation time steering the whole trajectory

Both generation-time interventions add to the residual stream at every layer of the token being written, at the launch token and at every generated token, and use the same leave-one-out donor pools and the same controls (norm-matched random direction, wrong value, mean over values, random labelling) as the suffix interventions. They differ only in how the direction is obtained.

Static: For each donor pair (x, y) in the pool, the program the model writes for x is teacher-forced through both prompts; the residual difference $h_\ell(x, \cdot) - h_\ell(y, \cdot)$ is averaged over the program's positions, then over pairs (and summed over factors for composition). The resulting per-layer vector is fixed for the cell and added with coefficient $\alpha = 0.25$, the best of a 0.0625–1.0 sweep. This is the standard steering-vector construction, read at the continuation positions.

Dynamic: The direction is recomputed at every generated token: each donor prompt in the pool is teacher-forced on the recipient's own output so far, $y_{<t}$, and the added vector is the pooled difference $\delta_{\ell,t} = \text{mean}_{(x,y)} \left[h_\ell(x, y_{<t}) - h_\ell(y, y_{<t}) \right]$, with $\alpha = 0.0625$ (0.125 for the strategy factor). Random controls are norm-matched to $\delta_{\ell,t}$ at each step. This is the same contrast the suffix intervention writes once into the K/V rows, delivered at the position where this model reads it.

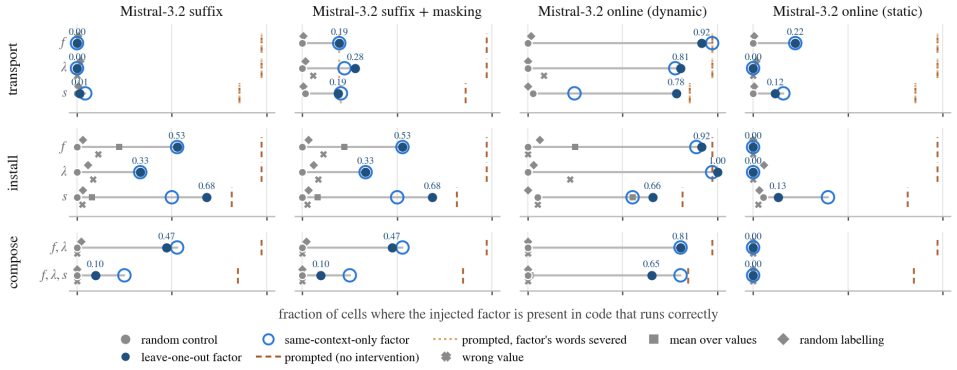

Figure 15: On Mistral-Small-3.2, the same intervention site that worked well on the main models largely fails.

Figure 15 shows the result of all 4 interventions. Note especially that the suffix-interventions that were largely successful in the main paper models are ineffective here; especially under transport. With no masking, the model sees the factor tokens in the prompt and the suffix steering is not enough to overwrite them. With masking, the intervention success increases but the overall coding performance plummets. The takeaway is that we cannot rule out abstractions in models that do not store reusable representations at their suffix; we simply must address them differently. They may sit somewhere between the suffix, the description tokens, already-generated tokens, or some mixture of all of the above.

A.12 What is the Compiled State?

We choose `Gemma-4-31B-it` for this investigation and provide additional clarification on what the "compiled" state at the prompt suffix is.

To investigate this, we ask the model to implement a function that sums the digits of an integer in Python. Additionally, we maintain a pool of 11 extra instructions that we add to the descriptions. These range from specifying the strategy (iteration vs. recursion) to requesting certain intermediary variable names or comments. We then consider 12 different increasing suffix lengths and for each, sample 10 trials of n different instructions from 1 to 10.

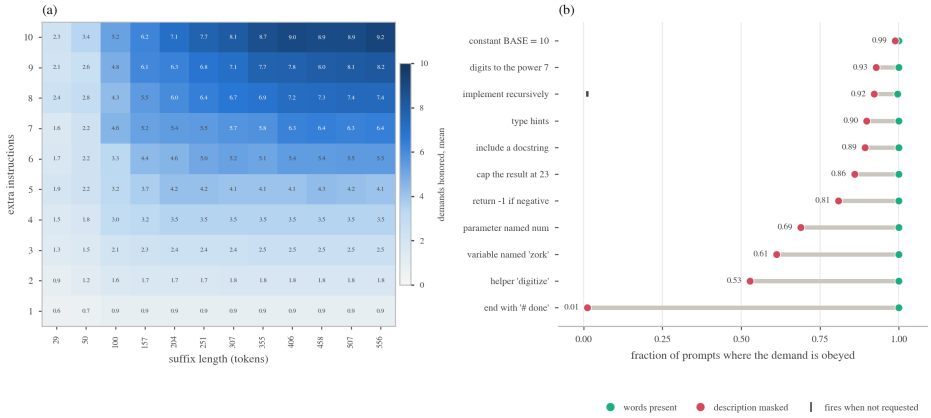

Figure 16: **(a)**: As more instructions are provided, a longer suffix length is needed for the model to store the program state. **(b)**: Not all instructions are equal. Instructions less relevant for computation are more often early re-read as opposed to being part of the computation state. With an unablated description, all instructions are honored.

Figure 16a shows that a larger number of extra instructions demands a longer suffix length to compile into a valid program abstraction. Additionally, we see in figure 16b that certain instructions generally compile less than others, and thus are more generally re-read as required rather than carried into the program state. For highly complex programs, it is clear that the model relies on a combination of its abstract program state and eagerly re-reading instructions.

A.12.1 Links to Reasoning

The construction in Appendix A.3 suggests a possible mechanism by which reasoning training could strengthen compiler-like internal representations.

The toy separates a fixed task condition $\theta = (f, \lambda)$ from its evolving machine states. A blank $\square$ leaves the function and watcher states unchanged, and its likelihood is independent of θ. For every supported extension $h\square^B$, the exact task belief $\eta_{\text{task}} = \eta_L \otimes \eta_F$ is therefore preserved:

$$\eta_{\text{task}}(h\square^B) = \eta_{\text{task}}(h).$$

This statement concerns Bayesian beliefs about the source, not equality of a trained model's activations. It motivates a model retaining task information across a region that supplies no further task evidence.

We hypothesize that reasoning tokens may create analogous pressure. They are not literally blank emissions: reasoning can perform substantial computation and introduce intermediate variables. However, many steps preserve the identity of the task being solved. A persistent activation state could therefore decompose schematically into a task contribution and an evolving workspace,

$$Z_t \approx Z_{\text{task}} \oplus Z_{\text{work},t},$$

with the task contribution retained while the workspace changes:

$$(Z_{\text{task}}, Z_{\text{work},t}) \longrightarrow (Z_{\text{task}}, Z_{\text{work},t+1}).$$

The hypothetical neural contribution Z_{task} need not be an exact encoding of η_{task}.

This suggests a possible origin for the compiler-like mechanism observed above. Long reasoning trajectories repeatedly require a model to preserve task constraints while performing many intervening computations. Reasoning training may therefore reward a division of labor in which the specification is "compiled" into a persistent latent state, while a separate state evolves to carry out the computation:

$$\text{specification} \longrightarrow \underbrace{Z_{\text{task}}}_{\text{persistent compiled state}} \longrightarrow \underbrace{Z_{\text{work},0} \to \cdots \to Z_{\text{work},T}}_{\text{reasoning computation}} \longrightarrow \text{realization.}$$

We emphasize that this is a hypothesis rather than a conclusion of the present experiments. The experiments here study models without reasoning; the possibility that factored representations persist or evolve throughout a reasoning trace is left open. This makes reasoning-trained models a particularly interesting setting in which to test whether extended computation strengthens the persistence, factorization, or causal transportability of compiled task state.

A direct prediction is that, conditional on the same task specification, reasoning-trained models should retain recoverable information about factors such as function, language, and strategy across longer stretches of intervening computation, including positions whose local token content contains little direct evidence about those factors. Note that this applies even to Transformer models with attention. As context length grows, it becomes harder for a few early tokens (for instance, initial instructions) to "win" the softmax and transport information. We do not claim that this is the only mechanism by which reasoning works, simply that it is one by which a compiled state may be computationally encouraged.

A.13 Strategy Steering

We expand on the results in figure 6 and for each model, extend this result.

For each language $\times$ language pair, we ask what a strategy donor from one language realizes in the other. Specifically we investigate how successful steering is from single donors and when it is, whether the steered realization matches prompting the donor, recipient, or something else, for cases where the recipient and donor disagree on the computation.

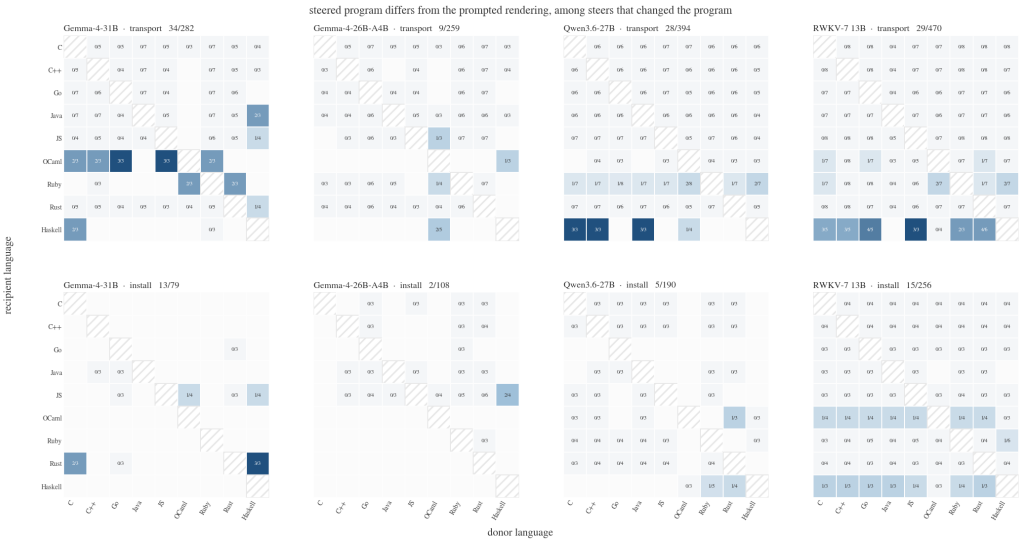

Figure 17: For each model pair and intervention, when strategy steering succeeds from a single donor, how is the resulting program expressed? Blank squares imply strategy steering did not work.

Donor program	Recipient after the donor state	Recipient asked in words
(a) Gemma-4-31B		

Haskell · digit_sum · iteration
`import Data.List (foldl')`
 	
`solve :: Integer -> Integer`
`solve n = foldl' (\acc d -> acc + toInteger`
`(fromEnum d - fromEnum '0')) 0 (show n)`
	Java · sq_digit_sum · Haskell state
	`public static int solve(int n) {`
	`return solveRecursive(n, 0);`
	`}`
	
	`private static int solveRecursive(int n, int`
	`acc) {`
	`if (n == 0) {`
	`return acc;`
	`}`
	`int digit = n % 10;`
	`return solveRecursive(n / 10, acc +`
	`(digit * digit));`
	`}`
		Java · sq_digit_sum · iteration in words
		`public int solve(int n) {`
		`int sum = 0;`
		`while (n > 0) {`
		`int digit = n % 10;`
		`sum += digit * digit;`
		`n /= 10;`
		`}`
		`return sum;`
		`}`

Rust · digit_sum · recursion
`fn solve(n: u64) -> u64 {`
`if n == 0 {`
`0`
`} else {`
`(n % 10) + solve(n / 10)`
`}`
`}`
	Haskell · sq_digit_sum · Rust state
	`solve :: Integer -> Integer`
	`solve n = if n == 0 then 0 else solve' n 0`
	`where`
	`solve' 0 acc = acc`
	`solve' x acc = solve' (x `div` 10) (acc +`
	`(x `mod` 10)^2)`
		Haskell · sq_digit_sum · recursion in words
		`solve :: Integer -> Integer`
		`solve 0 = 0`
		`solve n = (n `mod` 10)^2 + solve (n `div` 10)`

(b) Gemma-4-26B-A4B		

Haskell · digit_sum · recursion
`solve :: Integer -> Integer`
`solve 0 = 0`
`solve n = (n `mod` 10) + solve (n `div` 10)`
	JavaScript · sq_digit_sum · Haskell state
	`function solve(n) {`
	`return n.toString().split('').reduce((sum,`
	`digit) => sum + Math.pow(parseInt(digit),`
	`2), 0);`
	`}`
		JavaScript · sq_digit_sum · recursion in words
		`function solve(n) {`
		`if (n === 0) {`
		`return 0;`
		`}`
		`const digit = n % 10;`
		`return (digit * digit) + solve(Math.floor(n`
		`/ 10));`
		`}`

JavaScript · digit_sum · iteration
`function solve(n) {`
`let sum = 0;`
`let num = n;`
`while (num > 0) {`
`sum += num % 10;`
`num = Math.floor(num / 10);`
`}`
`return sum;`
`}`
	Ruby · digit_prod · JavaScript state
	`def solve(n)`
	`return n if n < 10`
	`product = 1`
	`digits = n.to_s.chars.map(&:to_i)`
	`digits.each { |d| product *= d }`
	`product`
	`end`
		Ruby · digit_prod · iteration in words
		`def solve(n)`
		`return 0 if n == 0`
		`product = 1`
		`while n > 0`
		`product *= n % 10`
		`n /= 10`
		`end`
		`product`
		`end`

(c) Qwen3.6-27B		

Haskell · digit_sum · recursion
`solve :: Int -> Int`
`solve n`
`| n < 10 = n`
`| otherwise = (n `mod` 10) + solve (n `div``
`10)`
	Ruby · digit_prod · Haskell state
	`def solve(n)`
	`return 0 if n == 0`
	`n.to_s.chars.map(&:to_i).reduce(1, :*)`
	`end`
		Ruby · digit_prod · recursion in words
		`def solve(n)`
		`return 1 if n == 0`
		`(n % 10) * solve(n / 10)`
		`end`

OCaml · digit_sum · recursion
`let rec solve n =`
`if n < 10 then n`
`else (n mod 10) + solve (n / 10)`
	Haskell · digit_prod · OCaml state
	`solve :: Integer -> Integer`
	`solve n = product $ map (\c -> read [c] ::`
	`Integer) $ show (abs n)`
		Haskell · digit_prod · recursion in words
		`solve :: Int -> Int`
		`solve n`
		`| n < 10 = n`
		`| otherwise = (n `mod` 10) * solve (n `div``
		`10)`

(d) RWKV-7 13B		

Haskell · digit_sum · iteration
`solve :: Integer -> Integer`
`solve n = go n 0`
`where`
`go 0 acc = acc`
`go x acc = go (x `div` 10) (acc + x `mod``
`10)`
	Ruby · digit_prod · Haskell state
	`def solve(n)`
	`product = 1`
	`while n > 0`
	`product *= n % 10`
	`n /= 10`
	`end`
	`product`
	`end`
		Ruby · digit_prod · iteration in words
		`def solve(n)`
		`product = 1`
		`n.to_s.each_char { |c| product *= c.to_i }`
		`product`
		`end`

Rust · digit_sum · iteration
`fn solve(n: u32) -> u32 {`
`let mut sum = 0;`
`let mut num = n;`
`while num > 0 {`
`sum += num % 10;`
`num /= 10;`
`}`
`sum`
`}`
	OCaml · even_digits · Rust state
	`let solve n =`
	`let rec count = function`
	`| 0 -> 0`
	`| n -> count (n / 10) + if n mod 10 mod 2`
	`= 0 then 1 else 0`
	`in`
	`count n`
		OCaml · even_digits · iteration in words
		`let solve n =`
		`let rec loop n count =`
		`if n = 0 then count`
		`else`
		`let digit = n mod 10 in`
		`let new_count = if digit mod 2 = 0 then`
		`count + 1 else count in`
		`loop (n / 10) new_count`
		`in`
		`loop n 0`

Figure 18: For each model, concrete examples where the steered recipient expresses a program differently from when it is prompted for the same strategy.

Figure 17 shows that successful strategy-steering from a single donor heavily depends on both the target and recipient language. Additionally, the steering sometimes causes realizations in forms that are different from how the recipient would normally write a given program if prompted to do it.

These results suggest that the interventions are not merely causing rewrites of the model prompts. Rather, when the models see a description containing a function and a language, the compiled state can also contain a language-specific intended realization of that computation. When we steer, this realization may sometimes carry over – more frequently when the languages differ substantially, as is the case with imperative vs. functional languages.

A.14 LOCAL GEOMETRY

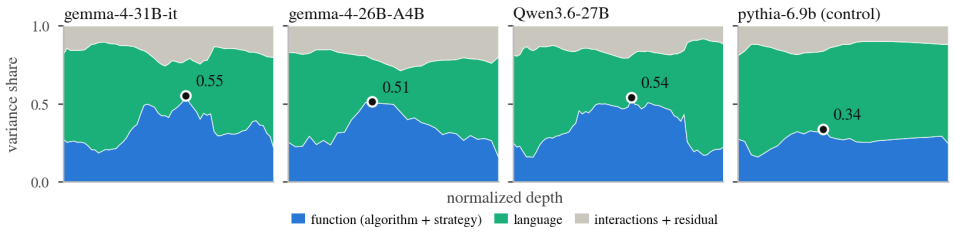

Figure 19: In the middle layers of the tested models, the function being implemented accounts for a larger share of the variance than the language it is implemented in.

There are two important consequences of the way we specify equation 2. First, no single layer has to realize the entire factorization. Different factors may be constructed and used at different depths. Second, if factors are reused, then at some point in the network variation in e.g. function identity should be attributable primarily to the function's computation rather than its lexical properties. We let each model solve a subset of 20 programs from our corpus in every language. We then stack and decompose those activations at every layer. Figure 19 shows the aforementioned consequences clearly; towards the middle of the model, computation-related main effects become a major source of variance, whereas language matters more before and after. Our control model shows signs of the same behavior, but its variance is consistently dominated by variation in language.

A.14.1 INTERVENTIONS

The interventions in the main paper show that function changes can be reused across contexts, but do not require function state to have globally linear coordinates. We now design interventions to probe the local geometry instead. First, let $Z(f)$ denote the measured state for function f with the remaining prompt context fixed.

We then ask whether a displacement estimated from one pair of functions can act as the same operation elsewhere. From a single donor pair we construct

$$\Delta_{\mathrm{sq}} = Z(\mathrm{sq} \cdot \mathrm{sum}) - Z(\mathrm{sum}), \tag{19}$$

and add it to the states for product and maximum. Neither $\mathrm{sq} \cdot \mathrm{prod}$ nor $\mathrm{sq} \cdot \max$ is used to construct the edit. We vary the dose α, decode in four programming languages, and compare against a norm-matched random perturbation. We run these experiments on `Gemma-4-31B`.

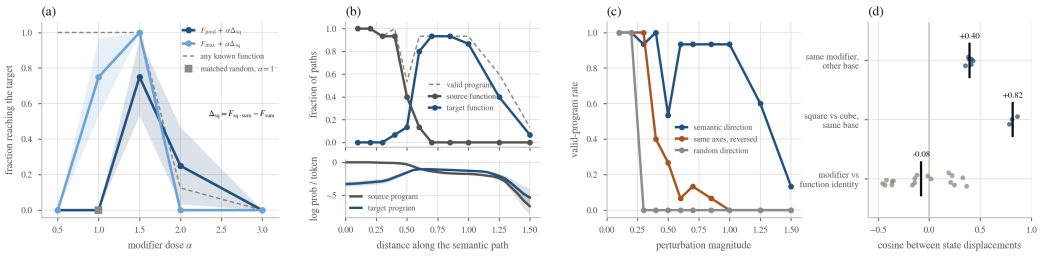

Figure 20: **Function state has locally reusable but anisotropic causal geometry.** **(a)** A square modifier estimated only from sum induces squared-product and squared-max at intermediate dose; a norm-matched random edit does not. **(b)** Moving along the displacement between two naturally elicited function states produces a sharp transition from source to target behavior, with validity falling near the transition and under over-extrapolation. **(c)** Valid computation is preserved much farther along the semantic direction than along its reversal or a matched random direction. **(d)** The same modifier across different bases is aligned, related modifiers on the same base are more strongly aligned, while modifier directions are approximately orthogonal to generic function differences. $F_{\mathrm{sum}}, F_{\mathrm{prod}}, F_{\max}$ denote the corresponding measured Z states.

Figure 20a shows that the same Δ_{sq} can produce both unseen modified functions, but only over a limited range of doses. This looks more like a reusable local operation than a global translation in function space.

We next move directly between two naturally occurring function states. For source f and target $f^\star$, we traverse

$$\widetilde{Z}(\alpha) = Z(f) + \alpha\,[Z(f^\star) - Z(f)]\,. \tag{20}$$

Here $\alpha = 0$ recovers the source and $\alpha = 1$ the same-context target state. Figure 20b shows a sharp transition between the two computations: near the boundary, validity drops, but as soon as it is traversed, valid programs are once again constructed.

We then compare the same semantic direction with its reversal and a norm-matched random direction orthogonal to it; directions that have *not* been shown to result in valid computation whatsoever. Figure 20c shows that this really matters. Traversing along the semantically-valid direction produces correct programs for a much higher degree of perturbation than the transverse and reverse. The picture is mixed: locally, computational directions can be used (and sometimes combined), but the effect is strongly dose-dependent. Arbitrary directions generally do not lead to valid programs.

A.14.2 Generalization of computational modifiers

We next ask whether this kind of modifier reuse extends beyond one donor pair and one model. We test the three selected language models and `RWKV7-13.3B` using several transformations, base computations, languages, and specification wordings. For RWKV, we intervene on its recurrent state rather than a KV-cache.

Let $M[f]$ be a transformation of computation f, and write $Z(f, \lambda, w)$ for the measured persistent state for function f, language λ, and wording w. For each recipient $c_\star = (f_\star, \lambda_\star, w_\star)$, we estimate a modifier only from donors with a different base function, language, and wording:

$$\widehat{\Delta}M^{(-c\star)} = \frac{1}{|\mathcal{D}_\star|} \sum_{(f,\lambda,w)\in\mathcal{D}_\star} [Z(M[f], \lambda, w) - Z(f, \lambda, w)]\,, \;\; \widetilde{Z}_\star \quad = Z(f_\star, \lambda_\star, w_\star) + \alpha\widehat{\Delta}M^{(-c\star)}. \tag{21}$$

Importantly, we use a leave-one-out design: the modifier does not contain the recipient function in modified form or examples from the recipient language.

We use sum, product, maximum, minimum, and alternating sum as digit aggregations and pick four modifiers: squaring the aggregate, aggregating squared digits, incrementing it, or doubling it. We use Python and JavaScript to select settings which we re-use for other languages. In total, there are 160 evaluation cells per model. Similarly to the main paper, success requires passing unit tests.

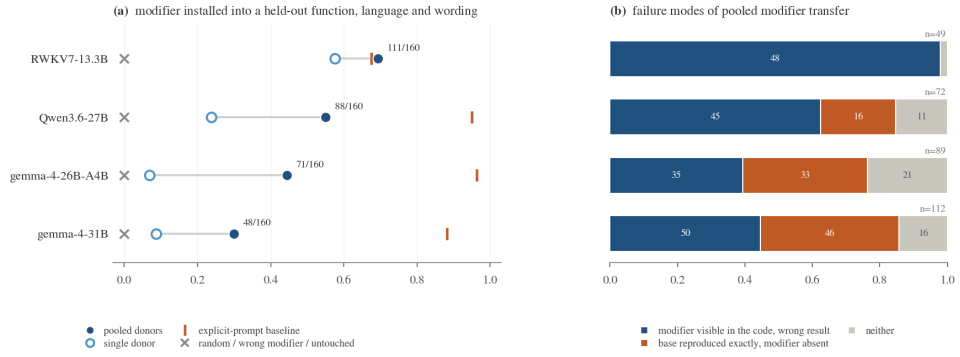

Figure 21: **Computational modifiers generalize across held-out contexts.** **(a)** Target-correct transfer across 160 cells per model. **(b)** Outcomes among unsuccessful pooled-donor interventions. Some failures still visibly express the modifier in its erroneous computation. Especially RWKV, which does so in 48 of its 49 misses.

Figure 21a shows modifier transfer in all four models. Pooled estimates succeed in 111/160 cells for `RWKV7-13.3B`, 88/160 for `Qwen3.6-27B`, 71/160 for `Gemma-4-26B-A4B`, and 48/160 for

`Gemma-4-31B`. Pooling improves over a single donor in every model, while none of the random, wrong-modifier, or unmodified controls produce the target computation.

Perhaps most interesting, an unsuccessful intervention does not always mean the intervention was inert. This is clearest for RWKV: of its 49 misses, 48 visibly express the modifier and none simply reproduce the unmodified base (Fig. 21b). Thus 159 of its 160 generations either implement the target or visibly express the intended transformation. The modifier arrives, the model just expresses it incorrectly.

There is a large difference in terms of how prompt information is available during generation. In transformers, the models can continue to use representations at earlier token positions, while RWKV must compress all of its information forward through its recurrent state.

Overall, the result is that computational modifiers *can* transfer across held-out functions, languages, wordings, and models, while remaining substantially more local than a globally linear function representation. However, the transfer also paints a complicated picture and reveals that we do not fully understand the geometry. An important caveat seems to be *where* in the program a certain modifier is expressed and *how* the model keeps track of this.

A.15 Steering Dosage

For `Gemma4-31B`, we additionally investigate how sensitive the interventions are to the choice of α; the scalar applied to our steering vectors.

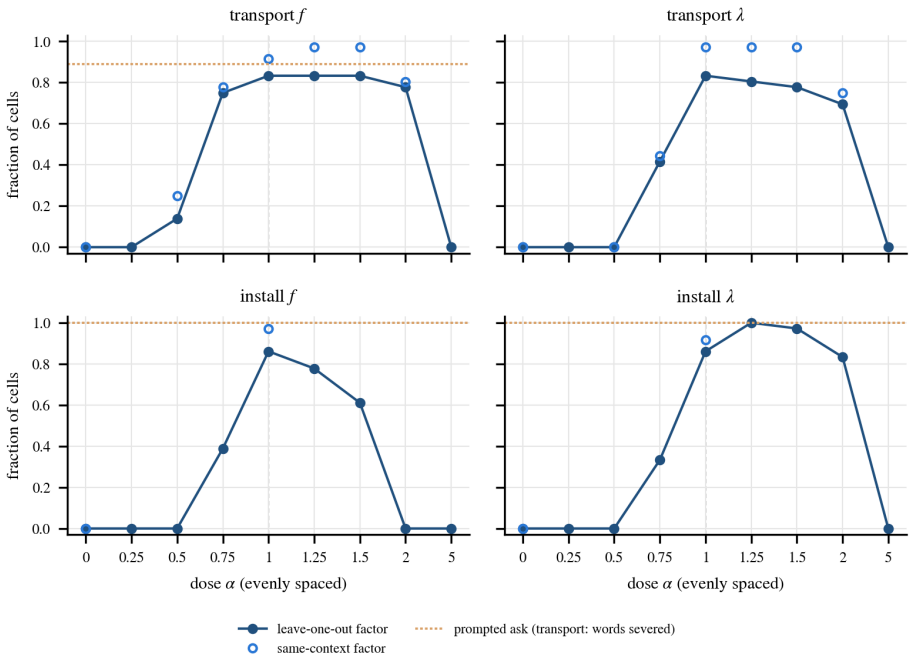

Figure 22: For language and function transport and install, how effective different α are.

Figure 22 shows that the intervention is fairly robust to choice of α, but that – as expected – performance drops off at more extreme values. Notably, values that are not exactly equal to 1 are still successful.

A.16 PARAPHRASES

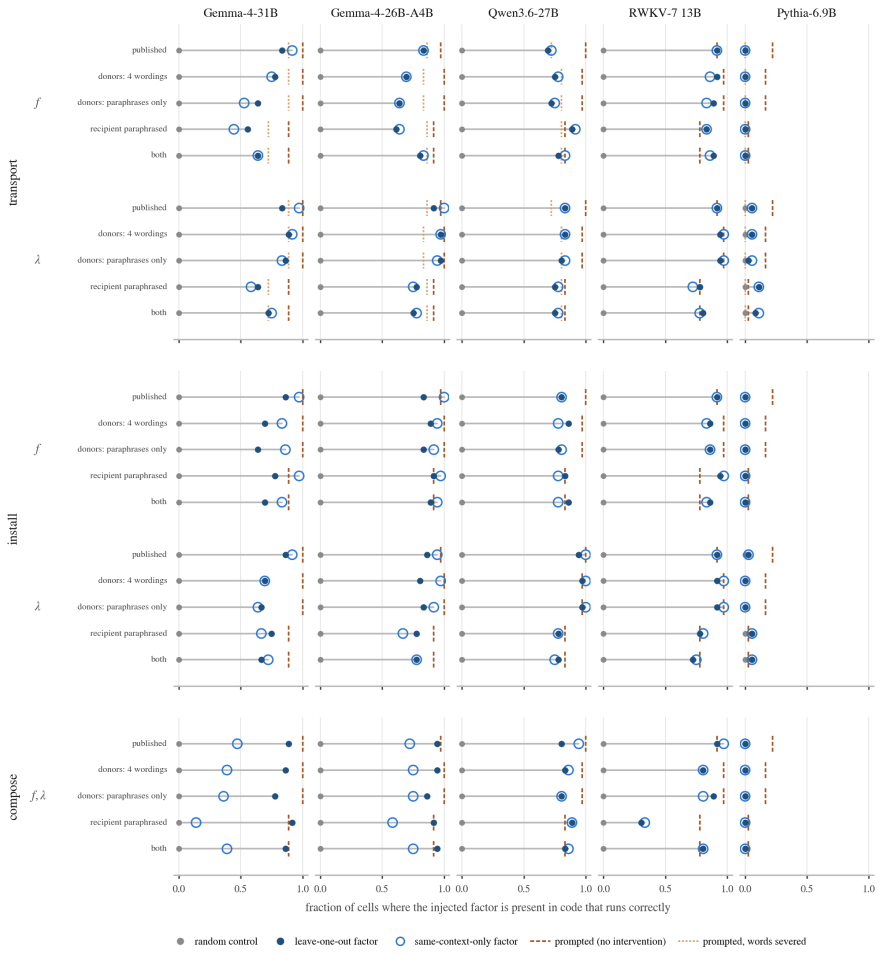

Figure 23: The paraphrasing interventions largely still succeed, albeit at a lower rate than when the descriptions are kept fixed between donor and recipient.

One objection to our experimental setup is that things are kept so similar between prompts that part of the effect can be attributed to being close to a context-swap. To address this, we investigate 4 additional conditions:

- Each donor is averaged over 4 different phrasings, the recipient is one of those.
- Donors randomly choose between 3 different phrasings, the recipient is a held-out phrasing.
- Donors have fixed phrasing, the recipient randomly chooses between 3 held-out phrasings.
- Each donor is averaged over 3 different phrasings, the recipient is a held-out phrasing.

Figure 23 shows that our interventions are largely robust to paraphrasing, but there is a notable decrease in performance, especially under the same-context interventions. These results argue that the interventions succeed not merely by transporting prompt-specific information between nearly identical contexts. The decrease under same-context interventions, however, suggests that the representation is not perfectly invariant to wording, where averaging appears to reduce this effect. A possible explanation is that averaging suppresses prompt-specific phrasings and isolates a more reusable representation. Future work should seek to understand this effect and determine if there is perhaps some shared causal subspace that averaging helps isolate.