Program-as-Weights (PAW): A Neural Compiler-Interpreter Framework for Fuzzy Functions

Wentao Zhang [*1] **Liliana Hotsko** [*1] **Woojeong Kim** [*2]
Pengyu Nie [1] **Stuart Shieber** [3] **Yuntian Deng** [1]

A wide range of everyday programming tasks resist clean, rule-based implementation: judging whether a review is positive, telling if an image contains an apple, or extracting a final answer from a chain-of-thought response with messy formatting and edge cases. Developers increasingly rely on large language model (LLM) APIs to handle such "fuzzy" logic, but doing so is expensive, internet-dependent, brittle to API changes, and prevents software from being self-contained. We introduce Program-as-Weights, a new programming paradigm that replaces symbolic code for fuzzy functions with compiled neural programs that run locally on a neural interpreter. A neural compiler translates high-level descriptions into compact neural programs (a combination of discrete tokens and continuous vectors) that, when interpreted by the neural interpreter, behave like black-box functions: they can be imported, called, versioned, and shared just like ordinary code. This enables local, lightweight, and reproducible execution without external LLM calls. We present initial prototypes on text and image processing tasks, and argue that Program-as-Weights opens the door to a new kind of software ecosystem—one where developers build and exchange libraries of neural functions alongside traditional code.

1. Introduction

Programming has historically been about writing explicit rules. A function is defined by code, and the computer executes it deterministically. For many tasks, this paradigm works beautifully: sorting numbers, processing structured data, or computing matrix multiplications. Yet a large class of real-world functions resist precise specification. Consider counting the number of verbs in a sentence, deciding whether an image contains an apple, or extracting a final numerical answer from a chain-of-thought response. Even ostensibly "simple" tasks, such as writing a regular expression to robustly parse text with many edge cases, often prove brittle or incomplete. Beyond underspecification, real-world inputs are noisy: typos, idiosyncratic punctuation, and format drift routinely break hand-written rules and regexes. These are fuzzy functions (Rubio Manzano, 2012)—problems that humans find intuitive but that cannot be fully captured by crisp symbolic rules.

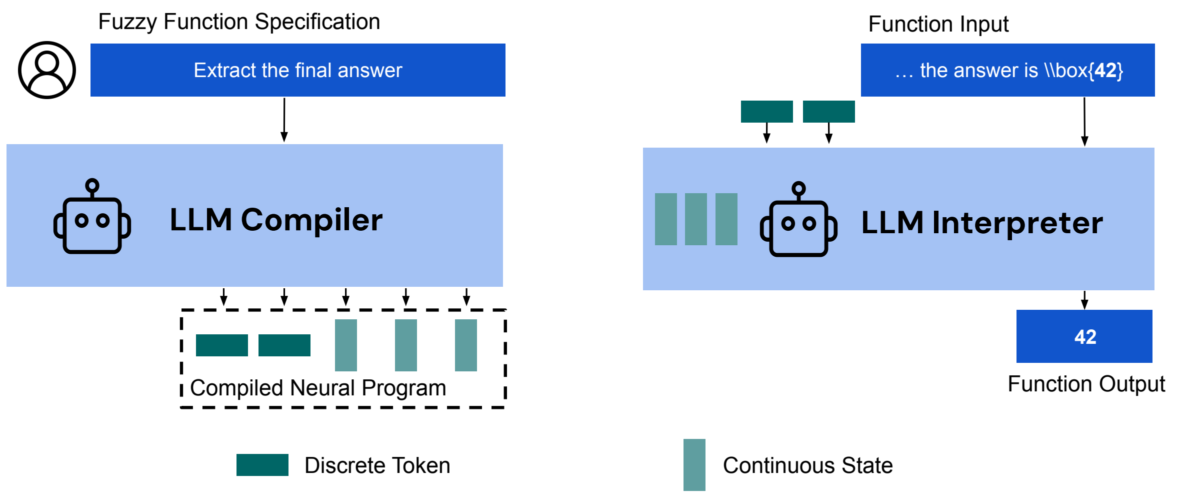
Today, developers frequently outsource such fuzziness to large language model (LLM) APIs. It is increasingly common to find codebases where a remote LLM is called (e.g., *gpt("extract answer", text)*) to implement functions that are otherwise intractable to program. This approach is undeniably convenient, but it comes with drawbacks. First, it is costly: calling a 10-billion-parameter model for every evaluation of a small utility function is excessive. Second, it is fragile: the function depends on internet connectivity and on the availability and reliability of a third-party service. Third, it undermines reproducibility: API providers may silently update or retrain their models, so the same function call may return different results over time (Kim et al., 2023). Finally, it prevents software from being self-contained: a project cannot be fully packaged or distributed if its correctness depends on an external API.

In this work, we propose a new programming paradigm: Program-as-Weights (PAW). The central idea is to replace symbolic code for fuzzy functions with compiled neural programs that can be executed locally by a neural interpreter. A neural compiler takes high-level specifications—natural language descriptions, input-output examples, or constraints—and produces neural programs. A neural program consists of a combination of discrete program tokens and continuous vectors, together encoding the logic required to implement the fuzzy function. A neural interpreter, available locally, executes this neural program just like a user-defined function. An overview of the paradigm is shown in Figure 1.

To realize this compiler-interpreter pipeline, we parameterize both components with transformer-based LLMs. The compiler is a large LLM trained to map specifications into compact parameterizations, while the interpreter is a smaller

*Equal contribution [1]Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada [2]Department of Computer Science, Cornell University New York, NY, USA [3]Harvard University, Cambridge, MA, USA. Correspondence to: Wentao Zhang <w564zhang@uwaterloo.ca>.

Proceedings of the 42nd International Conference on Machine Learning, Vancouver, Canada. PMLR 267, 2025. Copyright 2025 by the author(s).



(a) Compile (b) Interpret

Figure 1. **Overview of the Program-as-Weights paradigm**. A user specifies a fuzzy function (e.g., “extract the final answer”). The compiler compiles this specification into a compact neural weight blob that serves as the program. A fixed interpreter then executes the program locally as a callable function, enabling invocation through a simple Python API.

LLM that executes these parameterizations. In our implementation, programs are represented as discrete tokens and prefix vectors prepended to the interpreter’s input, serving as compiled “binaries” for fuzzy functions (Li & Liang, 2021). These neural binaries can be saved, version-controlled, and distributed alongside code, and the interpreter can load and execute them deterministically. We construct training datasets by synthesizing diverse specification-example pairs using LLMs and by repurposing existing datasets for the compiler-interpreter setting. We then train the full system end-to-end so that compiled programs generalize to new fuzzy functions beyond those seen during training.

We provide an initial prototype of this paradigm through programasweights.com, where users can write a function description, compile it into a neural program, and interactively test it. We also release a lightweight Python package, *programasweights* (*pip install programasweights*), that allows developers to compile and call fuzzy functions in just a few lines of code—running entirely locally, with no external model calls. In preliminary experiments on text and image processing and transformation, we find that fuzzy functions can indeed be realized as small weight blobs that are reproducible, efficient, and easy to integrate into existing software. All code and models can be found at anonymous.4open.science/r/programasweights.

Our contributions are as follows:

- **Paradigm:** We introduce neural fuzzy programming, where the program is defined not by strict rules but by neural programs to be executed by a neural interpreter.
- **Training:** We introduce a multi-stage training method to train the compiler-interpreter system.
- **System:** We implement a Python library, Program-As-Weights, that exposes both compiler and interpreter as practical tools.
- **Demonstration:** We provide experiments showing that this approach captures functions difficult to specify symbolically yet lightweight to run locally.
- **Vision:** We outline how this paradigm enables libraries of fuzzy functions that can be compiled once and reused widely, shifting how developers think about programming tasks that resist exact specification.

2. Programs as Weights

Let $f : X \to Y$ denote a function whose behavior is more naturally specified through natural language, examples, or constraints, rather than symbolic code. Instead of repeatedly invoking LLMs to approximate such behavior, we propose neural programs: compact weight blobs that specialize a fixed model to implement the fuzzy function f.

Formally, let S denote a user specification, expressed in natural language and optionally accompanied by example input-output pairs (x, y) that illustrate the desired behavior of f. A neural `Compiler` maps the specification s to a continuous vector p representing the compiled program:

$$\texttt{Compiler} : s \to p.$$

A neural interpreter `Interpreter` executes the program by applying these weights p to an input $x \in X$ to produce an output $y \in Y$:

$$f(x) \approx \texttt{Interpreter}(\texttt{Compiler}(s), x).$$

This division mirrors the classical structure of programming, where a compiler translates source code into a binary that is executed by an interpreter or runtime. The key difference in our framework is that the binary is a learned parameter vector, and the interpreter is a neural network.

3. The Compiler-Interpreter System

We instantiate the Program-as-Weights paradigm with a concrete compiler-interpreter architecture composed of two transformer models: a *neural compiler*, which produces a neural program, and a *neural interpreter*, which executes it. Given a specification, the compiler generates a compact program representation P, and the interpreter conditions on p to realize the corresponding fuzzy function.

Neural Program For conceptual simplicity, we described a neural program as a single continuous object p. In practice, however, we represent it as a hybrid of a discrete and a continuous component:

$$p = \left(p_{\text{discrete}}, p_{\text{continuous}}\right).$$

The discrete component p_{discrete} is a variable-length sequence of tokens that acts as a lightweight "pseudo-program," while the continuous component $p_{\text{continuous}}$ consists of a fixed number of continuous vectors that capture behaviors that are difficult to express symbolically.

At execution time, both components are provided to the interpreter as part of its context. Importantly, this mixed representation is still a special case of a purely continuous program. The discrete tokens are consumed by the interpreter through its embedding and hidden layers, after which they participate in attention in the same way as any other continuous representation. From the interpreter's perspective, the embedded discrete tokens and the continuous vectors together form a single block of continuous activations. Thus, without loss of generality, a neural program can be viewed as a unified continuous representation.

Neural Compiler Given a specification s, the compiler produces a program p. The compiler is a transformer language model that first autoregressively generates the discrete pseudo-program p_{discrete}, terminating either at an end-of-sequence token or a maximum generation length.

To generate the continuous component $p_{\text{continuous}}$, we append N learned "prefix tokens" to the end of the compiler's input. After a forward pass, the hidden states corresponding to these prefix tokens are extracted and formatted as a key-value (KV) prefix representation. To ensure compatibility with the interpreter, we extract the KV cache from the compiler, and then use an MLP layer to project them to the same size as the hidden size of the interpreter. This produces a program representation that can be directly injected into the interpreter's attention mechanism.

Neural Interpreter The neural interpreter is a smaller transformer that executes the compiled program. At inference time, the continuous component $p_{\text{continuous}}$ is injected into the interpreter as KV prefix states, while the discrete pseudo-program p_{discrete} is prepended to the interpreter's input sequence. Together, these components condition the interpreter throughout generation.

Given an input x, the interpreter autoregressively produces an output

$$\hat{y} = \texttt{Interpreter}(p_{\text{continuous}}, p_{\text{discrete}}, x).$$

Because the interpreter remains frozen, new fuzzy functions can be introduced simply by loading different programs p. This allows neural "binaries" to be reused, shared, and composed without retraining the interpreter itself.

Multimodal Inputs The compiler-interpreter framework naturally extends to multimodal settings. The compiler can accept image inputs via a frozen vision encoder (e.g., CLIP (Radford et al., 2021)), which converts an image into a sequence of embeddings projected into the transformer embedding space. These embeddings are treated identically to text tokens, allowing text and image inputs to be freely interleaved. This enables multimodal specifications for the compiler and multimodal inputs for the interpreter. We leave the generation of image outputs to future work.

4. Training

Training data consist of triplets (s, x, y), where s is a natural language specification of a fuzzy function, x is an input to that function, and y is the target output.

Objective We train the compiler and interpreter jointly to maximize the likelihood of producing the correct function behavior,

$$\log P(y \mid s, x).$$

In practice, only the compiler parameters θ are optimized, while the interpreter (with parameters ϕ) is kept frozen. Given a specification s, the compiler samples a neural program p, which conditions the interpreter to produce an output distribution $P_\phi(y \mid p, x)$.

A central challenge is that we do not observe ground-truth discrete pseudo-programs p_{discrete}. Indeed, discovering effective pseudo-programs is part of the learning problem itself. We therefore treat program generation as a latent decision process and optimize the compiler using policy gradient (Sutton et al., 1999). The resulting objective can be written as

$$-\mathbb{E}_{p\sim P_\theta(p|s)}\left[\log P_\theta(p \mid s)\,\texttt{stopgrad}(r(p,x)) + r(p,x)\right],$$

where the reward

$$r(p,x) = -\log P_\phi(y \mid p, x)$$

is the negative log-likelihood of the interpreter producing the correct output. Here, ϕ denotes frozen interpreter parameters.[1]

Directly optimizing this joint objective with online RL is unstable and inefficient: it requires interleaving rollout generation and optimization, which imposes conflicting constraints on batch size, policy freshness, and hardware utilization. To address this, we adopt a staged training strategy that progressively introduces complexity while maintaining stability and efficiency.

Stage 1: RL training for discrete pseudo-programs In the first stage, we remove the continuous component of the program and train the compiler to generate effective discrete pseudo-programs p_{discrete} only. Intuitively, this stage teaches the compiler to act as a prompt engineer for the interpreter (Zhou et al., 2023).

The objective reduces to a policy gradient loss over discrete programs:

$$-\mathbb{E}_{p_{\text{discrete}}\sim P_\theta(\cdot|s)}\left[\log P_\theta(p \mid s)\,\texttt{stopgrad}(r(p_{\text{discrete}},x))\right].$$

In practice, we subtract a baseline from the reward to reduce variance. The baseline is the average reward across a group of rollouts sampled for the same (s,x) pair, following the Group Relative Policy Optimization (GRPO) paradigm (Shao et al., 2024; Guo et al., 2025).

After this stage, the compiler produces discrete pseudo-programs that meaningfully condition the interpreter. However, we find that discrete programs alone often provide insufficient control when the interpreter is small, leading to errors such as incorrect formatting. This motivates the introduction of a continuous program component.

Stage 2: Supervised training for continuous programs In the second stage, we train the compiler to generate the continuous component $p_{\text{continuous}}$, which provides better control over the interpreter's behavior. To avoid the inefficiencies of online RL, we proceed offline.

[1]While recent RL-based LLM training often uses exact-match rewards, we found such signals to be extremely sparse for our tasks. In contrast, log-likelihood-based rewards and negative edit distance provide smoother and more informative learning signals, and both work well in practice.

Specifically, for each specification s, we sample and store K discrete pseudo-programs p_{discrete} produced by the Stage 1 compiler. Treating these discrete programs as fixed, we optimize the continuous program generation via supervised learning:

$$-\mathbb{E}_{p_{\text{discrete}}}\left[r(p_{\text{continuous}}, p_{\text{discrete}}, x)\right].$$

Because the discrete component is held fixed, this objective reduces to standard supervised fine-tuning (SFT) that maximizes the interpreter's likelihood of producing the target output y.

A potential issue is distributional drift: updating the compiler may cause it to no longer generate the same discrete pseudo-programs used during offline training. To mitigate this, we introduce a regularization term that encourages the compiler to remain close to the original policy:

$$-\mathbb{E}_{p_{\text{discrete}}}\left[r(p_{\text{continuous}}, p_{\text{discrete}}, x) + \lambda \log P_\theta(p_{\text{discrete}} \mid s)\right],$$

where $\lambda > 0$ controls the strength of the regularization.

Stage 3: Joint RL and SFT refinement Despite regularization, some mismatch between discrete and continuous program generation may remain. We therefore perform a final joint refinement stage using the original combined objective:

$$-\mathbb{E}_{p\sim P_\theta(p|s)}\left[\log P_\theta(p \mid s)\,\texttt{stopgrad}(r(p,x)) + r(p,x)\right].$$

At this point, both the compiler and the induced interpreter behavior are well initialized, so only a small number of additional gradient steps are required.

In practice, we follow standard large-scale RL training conventions and use slightly stale policies (Zeng et al., 2025): rollouts are generated in batches using a fixed compiler snapshot, treated as on-policy until exhausted, and then refreshed using the latest compiler parameters. This approach strikes a favorable balance between stability, efficiency, and final performance.

5. System Design

A central contribution of Program-as-Weights is a practical and developer-friendly workflow. We design the system so that fuzzy functions can be *compiled once* into self-contained neural programs and subsequently *executed anywhere* on a fixed interpreter. This section describes how developers create, test, and deploy PAW functions in practice.

5.1. Python API: Compile and Run

Developers interact with PAW through a simple two-step workflow: *compile*, then *call*. Figure 2 shows a complete

```
import programasweights as paw

# (1) Describe the fuzzy function
prompt = "Parse a string like \
    '(A) ... (B) ... (C) ...' into \
    a JSON dictionary that \
    maps each option label to \
    its corresponding option text."

# (2) Compile to a neural program
f = paw.compile(prompt=prompt)
f.dump("parser.weights")
```

Listing 1. Compile a fuzzy function into a weight blob

```
import programasweights as paw

# (3) Load the program blob
f = paw.function("parser.weights")

# (4) Call it like a function
s = "(A) cat  (B) dog
(C) both (A) and (B)"

print(f(s))
# -> {"A":"cat","B":"dog", \
#     "C":"both (A) and (B)"}
```

Listing 2. Run the compiled program locally via the interpreter

Figure 2. **Program-as-Weights developer interface.** *Left:* The compiler translates a natural language specification into a compact neural weight blob (the "neural program"). *Right:* The interpreter loads this program and exposes it as a callable function, enabling local, deterministic execution without reliance on external LLM APIs.

minimal example. The developer begins by specifying a fuzzy function in natural language. The compiler translates this specification into a compact neural program, which is serialized as a program blob. The neural interpreter then loads this blob and exposes it as a standard Python callable.

This workflow requires no familiarity with machine learning internals such as attention mechanisms, prefix vectors, or neural representations. From the developer's perspective, neural programs behave like ordinary functions, except that they can encode fuzzy or underspecified behavior that is brittle or impractical to express using symbolic code.

Crucially, once compiled, a neural program executes locally without invoking any external LLM APIs. This design ensures deterministic execution, reproducibility, and full offline functionality at deployment time.

5.2. Web Compiler Interface

To further lower the barrier to adoption, we provide a hosted web-based compiler interface at programasweights.com. The web compiler allows users to:

1. Enter a program specification in natural language;
2. Test the resulting neural program on multiple inputs;
3. Inspect, refine, and validate its behavior;
4. Download the compiled program blob once satisfied.

Compilation is performed on GPU-backed servers, enabling the use of more powerful compiler models than would typically be available on a user's local machine. Once downloaded, the resulting program executes entirely offline on the local interpreter. Alternatively, PAW supports optional on-demand fetching: providing only a program identifier automatically retrieves the corresponding program from the server on first use.

This design cleanly separates development from deployment. Developers iterate and debug programs in the cloud, while end users deploy lightweight neural binaries as ordinary code artifacts. Neural programs can be version-controlled, stored in repositories, or shipped alongside software dependencies. Because all programs execute on a single fixed interpreter, introducing new fuzzy functions requires no additional training or fine-tuning.

Local Runtime Requirements At deployment time, users only need to install the PAW runtime and a single fixed neural interpreter model. In all experiments, we use a Qwen3-0.6B interpreter, which occupies approximately 1.2 GB of disk space in bfloat16 precision. This interpreter is shared across all PAW programs and installed once.

Each compiled neural program is distributed as a self-contained program blob, typically on the order of a few megabytes (see Table 1 for size statistics). No compiler model, training code, or GPU is required at runtime.

6. Experimental Setup

We evaluate Program-as-Weights (PAW) on two classes of problems: text processing tasks and image processing tasks. Together, these benchmarks test PAW's ability to implement fuzzy functions over both unimodal and multimodal inputs.

6.1. Text processing Tasks

To study fuzzy function implementation in the text domain, we construct a benchmark suite that we call *FuzzyBench*. FuzzyBench is designed to evaluate cross-task generalization: the model must translate previously unseen natural language specifications into executable fuzzy functions, rather than overfitting to a fixed task.

The dataset is constructed via prompting a large language model (gpt-5-mini) in two stages:

1. We first prompt the model to generate natural language specifications of fuzzy functions.
2. Given each specification, we then prompt the model to generate multiple input-output pairs that instantiate the specified behavior.

Using this procedure, we generate 81,920 non-duplicate specifications. For each specification, we sample 8 input-output examples, yielding 655,360 total examples. After filtering 3,060 ill-formed cases, the final dataset contains 652,300 examples. Specifications are split into 80% training, 10% validation, and 10% test sets, ensuring that specifications in the test set are never observed during training.

The exact prompts used for dataset construction are provided in Section B. As an example:

- Specification: Extract the final answer from a reasoning trace.
- Input: Final answer `\boxed{2}`.
- Output: 2.

In addition to FuzzyBench, we evaluate PAW on four standard text classification datasets following ALCHEmist (Huang et al., 2024). We use the four datasets drawn from the WRENCH benchmark (Zhang et al., 2021): YouTube, SMS, Yelp, and IMDb.

6.2. Image processing Tasks

We also evaluate PAW on image processing tasks, where fuzzy functions are conditioned on visual inputs. We formulate these problems as image-conditioned question answering: the compiler receives an image as part of the specification, and the interpreter answers queries about the image.

As a motivating example, one may view a scanned research document as being compiled into a fuzzy function that supports question answering over its contents. More broadly, this setup evaluates whether PAW can translate multimodal specifications into executable neural programs.

We evaluate on a range of OCR and visual reasoning benchmarks, including Im2LaTeX (formula image to LaTeX conversion) (Deng et al., 2017), Im2Smiles (molecule image to SMILES conversion) (Deng et al., 2023), and visual question answering tasks over chemical structures, electronic circuits, and musical scores from the CoSyn-400k benchmark (Yang et al., 2025b; Deitke et al., 2024).

6.3. Models and Baselines

Text processing For text processing experiments, we use Qwen3-4B (Yang et al., 2025a) as the compiler and Qwen3-0.6B as the interpreter.

As baselines, we compare against:

- Prompting gpt-5-mini directly, which serves as an approximate upper bound on FuzzyBench performance, since the same model is used for data generation;
- Prompting open-weight local LLMs, including Qwen3-0.6B, Qwen3-1.7B, Qwen3-4B, Qwen3-8B, and OLMo3-7B (Olmo et al., 2025);
- The program synthesis baseline introduced in ALCHEmist (Huang et al., 2024), which first prompts an LLM to generate a Python program and then executes it using a Python interpreter. We report both numbers from the original paper and our reproduced results.

Image processing For image tasks, we use Qwen3-VL-4B as the compiler and the same Qwen3-0.6B interpreter as in text experiments. Baselines include prompting open-weight vision-language models AndesVL-0.6B (AndesVL Team, 2025), Qwen3-VL-2B, and Qwen3-VL-4B.

6.4. Implementation Details

For text processing tasks, we initialize the compiler with prompts that encourage it to generate structured pseudo-programs.[2] Full prompts are provided in Section C. We constrain the pseudo-programs to be at most 256 tokens, and we use 64 prefix tokens to extract the continuous KV cache unless otherwise stated.

During training, we cap the total sequence length at 950 tokens for both the compiler and interpreter. In Stage 1 training, we used fresh on-policy rollouts with a batch size of 1 and 32 rollouts per example. We use a learning rate of 1×10^{-6} and full FP32 precision for stability. Due to memory constraints, Stage 1 training is performed on a single NVIDIA B300 GPU.

In Stage 2, we use a regularization strength of 0.1, a batch size of 24, gradient accumulation of 2, bfloat16 precision, and a learning rate of 2×10^{-5}. In Stage 3, we generate rollouts using vLLM for 1,024 examples with 8 rollouts per example, and train with a batch size of 2×8, gradient accumulation of 16, and a learning rate of 5×10^{-7}.

For image processing tasks, we use prompts in Section D. The discrete pseudo-program length is capped at 256. We set the maximum sequence length to 4,096 tokens for the compiler and 1,040 tokens for the interpreter.

In Stage 1, we perform RL training on 8 NVIDIA H200s with a batch size of 32 and a learning rate of 1×10^{-6}. In Stage 2, we apply LoRA fine-tuning with rank $r = 32$. We

[2] In preliminary experiments, we observed that the compiler naturally learns to emit paraphrased specifications followed by as many input-output examples as allowed by the generation budget. We incorporate this inductive bias into the initial prompts, which substantially accelerates RL training.

Table 1. Results on text processing tasks. *Contained* indicates whether the program is self-contained and executable without internet access. PS denotes program size. †: numbers taken from Huang et al. (2024). *: 0 due to zero recall. For the LLM→Code baseline, the reported 29 MB corresponds to the size of the Python 3.14.2 package (pyt, 2025).

Model	Contained	Interp. Size	FuzzyBench		YouTube		SMS		Yelp		IMDB	
			Acc	PS	Acc	PS	F1	PS	Acc	PS	Acc	PS
GPT-5-mini (API)	×	-	57.45%	0.73 KB	93.60%	0.95 KB	91.03%	1.03 KB	98.13%	1.69 KB	94.96%	2.25 KB
Local LLM (Qwen3 0.6b)	✓	0.6b	17.26%	0.73 KB	52.80%	0.95 KB	0.00%*	1.03 KB	89.55%	1.69 KB	66.88%	2.25 KB
Local LLM (Qwen3 1.7b)	✓	1.7b	28.88%	0.73 KB	88.40%	0.95 KB	71.43%	1.03 KB	84.39%	1.69 KB	59.12%	2.25 KB
Local LLM (Qwen3 4b)	✓	4b	43.46%	0.73 KB	90.80%	0.95 KB	92.54%	1.03 KB	97.53%	1.69 KB	93.76%	2.25 KB
Local LLM (Qwen3 8b)	✓	8b	44.38%	0.73 KB	94.40%	0.95 KB	91.55%	1.03 KB	97.95%	1.69 KB	94.52%	2.25 KB
Local LLM (Olmo3 7b)	✓	7b	37.70%	0.73 KB	90.00%	0.95 KB	90.14%	1.03 KB	97.66%	1.69 KB	93.28%	2.25 KB
LLM→Code (Huang et al., 2024)	✓	29 MB	-	-	89.1%†	-	90.0%†	-	57.5%†	-	66.2%†	-
LLM→Code (Reimplementation)	✓	29 MB	35.81%	0.08 KB	70.46%	0.09 KB	86.41%	0.06 KB	50.35%	0.05 KB	73.92%	0.08 KB
PAW	✓	0.6b	44.21%	7.17 MB	95.20%	7.17 MB	96.35%	7.17 MB	97.34%	7.17 MB	94.08%	7.17 MB

Table 2. Results on image processing tasks.

Model	Contained	Interp. Size	Chemical		Circuit		Music		Im2Smiles		Im2LaTeX	
			Acc	PS	Acc	PS	F1	PS	Acc	PS	Acc	PS
Local LLM (AndesVL 0.6b)	✓	0.6b	18.27%	1.14 MB	21.41%	1.25 MB	44.80%	2.21 MB	0.00%	0.19 MB	43.47%	0.06 MB
Local LLM (Qwen3-VL-2b-Instruct)	✓	2b	25.77%	1.14 MB	18.64%	1.25 MB	46.97%	2.21 MB	1.60%	0.19 MB	40.76%	0.06 MB
Local LLM (Qwen3-VL-4b-Instruct)	✓	4b	22.05%	1.14 MB	19.61%	1.25 MB	45.03%	2.21 MB	4.40%	0.19 MB	39.86%	0.06 MB
PAW	✓	0.6b	36.53%	7.17 MB	24.10%	7.17 MB	52.46%	7.17 MB	17.50%	7.17 MB	39.06%	7.17 MB

use a learning rate of 1×10^{-4}, a regularization strength of 0.1, and a batch size of 48. This stage is conducted on 4 NVIDIA H100 GPUs. Unlike the text processing pipeline, we do not perform Stage 3 joint training for image tasks.

7. Results and Analysis

7.1. Main Results

Table 1 shows the text processing results. Despite using the *same* interpreter as the Local LLM baseline (Qwen3-0.6B), PAW substantially improves performance by introducing a compiler. In effect, PAW transforms a small, fixed-capacity interpreter into a task-adaptive model through compilation.

Notably, PAW achieves performance comparable to directly prompting a local LLM with 8B parameters, while relying on an interpreter with only 0.6B parameters. This shows the effectiveness of compiling task-specific structure into neural programs rather than scaling the execution model itself.

PAW also significantly outperforms the program synthesis baseline that first generates Python code and then executes it. This gap reflects a fundamental limitation of symbolic programs for fuzzy tasks: hand-generated code struggles to capture linguistic variability, implicit conventions, and noisy real-world inputs. For example, on the Yelp dataset, synthesized Python functions achieve only 57.5% accuracy—barely above chance—while all neural approaches handle the task reliably. In practice, even minor idiosyncrasies such as hyphenation or plus-sign variants can cause heuristic rules to either overmatch or miss entirely.

There is, however, no free lunch. PAW programs are substantially larger than traditional Python scripts: the average neural program is 7.17 MB, compared to 0.05-0.09 KB for Python code. This difference reflects where complexity is stored. Fuzzy functions inherently require knowledge of world regularities, linguistic variation, formatting noise, and numerous exception cases, details that are tedious for humans to enumerate. Neural programs succeed because they encode far more information than typical handwritten code. PAW does not make fuzzy functionality *simple*; rather, it automates, compresses, and externalizes this complexity, shifting the burden from developers to the compiler.

Table 2 summarizes results on image processing. Consistent with the text domain, PAW substantially improves the performance of comparably sized local models, further validating the effectiveness of using a neural compiler to adapt a fixed, lightweight interpreter to complex multimodal tasks.

7.2. Ablation Studies

Contribution of each training stage Table 3 reports validation accuracy on FuzzyBench after each training stage. The initial prompting-based system achieves only 9.67% accuracy. This is worse than directly prompting the interpreter, likely due to error compounding from first prompting the compiler and then using its output to prompt the interpreter.

After Stage 1 RL training, the compiler learns to generate more effective discrete pseudo-programs, improving accuracy to 19.53%. Empirically, we observe that regardless of initialization, the compiler converges to emitting a

Table 3. Validation accuracy after each training stage.

Stage	Accuracy
No Training	9.67%
Stage 1 (RL)	19.53%
Stage 2 (SFT)	42.07%
Stage 3 (Joint RL+SFT)	45.68%

Table 4. Effect of program representations.

Variant	Accuracy
Discrete Only (RL trained)	19.53%
Continuous Only (SFT trained)	22.73%
Full Model	45.68%

paraphrased specification followed by as many input-output examples as permitted by the token budget.

Stage 2 yields the largest improvement: after supervised training of the continuous program component, accuracy increases to 42.07%. This indicates that continuous program representations provide substantially finer control over the interpreter than discrete prompts alone. Finally, Stage 3 joint training further improves accuracy to 45.68%.

Discrete vs. continuous programs Table 4 compares different program representations. Using a hybrid of discrete and continuous components outperforms using either component alone.

Effect of discrete program length During Stage 1 training, we observe that the compiler consistently exhausts the available discrete token budget. This raises the question of whether the discrete program length limits performance. Table 5 varies the maximum number of tokens allowed in the discrete pseudo-program. Performance improves as the discrete program length increases. However, scaling beyond 256 tokens leads to out-of-memory (OOM) errors in our current setup, preventing further exploration.

Effect of continuous program length Table 6 evaluates the impact of the number of continuous prefix tokens used to construct the KV cache. Increasing the continuous program length yields modest gains up to 64 tokens, after which improvements plateau while program size continues to grow.

Effect of compiler size Table 7 shows that larger compilers lead to better performance. While this suggests favorable scaling behavior, we limit experiments to compilers up to 4B due to resource constraints. Exploring compiler scaling laws remains an important direction for future work.

Table 5. Effect of discrete pseudo-program length.

Max Discrete Program Length	Accuracy
64	26.78%
128	36.45%
256	45.68%

Table 6. Effect of continuous program length (# prefix tokens).

# Prefix Tokens	Accuracy	PS
16	38.21%	1.79 MB
32	38.26%	3.59 MB
64	45.68%	7.17 MB
128	46.00%	14.34 MB

Table 7. Effect of compiler model size.

Compiler Size	Accuracy
0.6B	16.99%
1.7B	29.00%
4B	45.68%

8. Conclusion

We introduced Program-as-Weights, a programming paradigm in which neural weights function as modular, callable programs for implementing fuzzy functions. By decoupling a powerful neural compiler from a lightweight, fixed neural interpreter, PAW allows developers to express intent at a high level while executing the resulting functionality locally, efficiently, and reproducibly. Our experiments demonstrate that these neural binaries can implement behaviors that are difficult to encode symbolically, while avoiding dependence on remote LLM APIs.

More broadly, Program-as-Weights points toward a software ecosystem in which neural binaries implementing fuzzy logic can be versioned and shared much like conventional libraries, enabling symbolic and neural programming to coevolve rather than compete. By shifting complexity into compilation while keeping inference local and lightweight, PAW offers a concrete realization of the long-anticipated "small LM future" (Belcak et al., 2025). We believe this paradigm invites a rethinking of what it means to write and deploy software in the age of intelligent computation.

Impact Statement

Program-as-Weights aims to simplify the deployment and maintenance of machine learning-enabled software by reducing dependence on internet connectivity and third-party services, and by enabling reproducible, version-controlled distribution of self-contained neural programs. By separating development from execution, the approach may lower operational barriers for deploying learned systems in constrained or offline environments.

As with other methods that reduce the cost of using learned models, Program-as-Weights may also broaden access to machine learning–based functionality and facilitate its integration into a wider range of applications. We do not anticipate risks or societal impacts unique to this paradigm beyond those commonly associated with increased accessibility of machine learning technologies, such as potential misuse or over-reliance on automated systems. These considerations are not specific to Program-as-Weights and are best addressed through established practices in responsible model development, evaluation, and deployment.

References

Python 3.14.2. https://www.python.org/downloads/release/python-3142/, 2025. Release date: December 5, 2025.

AndesVL Team, O. A. C. Andesvl technical report: An efficient mobile-side multimodal large language model, 2025. URL https://arxiv.org/abs/2510.11496.

Belcak, P., Heinrich, G., Diao, S., Fu, Y., Dong, X., Muralidharan, S., Lin, Y. C., and Molchanov, P. Small language models are the future of agentic ai, 2025. URL https://arxiv.org/abs/2506.02153.

Charakorn, R., Cetin, E., Tang, Y., and Lange, R. T. Text-to-loRA: Instant transformer adaption. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=zWskCdu3QA.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Chen, T., Fang, H., Xia, P., Liu, X., Durme, B. V., Zettlemoyer, L., Gao, J., and Cheng, H. Generative adapter: Contextualizing language models in parameters with a single forward pass. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=bc3sUsS6ck.

Deitke, M., Clark, C., Lee, S., Tripathi, R., Yang, Y., Park, J. S., Salehi, M., Muennighoff, N., Lo, K., Soldaini, L., et al. Molmo and pixmo: Open weights and open data for state-of-the-art multimodal models. *arXiv preprint arXiv:2409.17146*, 2024.

Deng, Y., Kanervisto, A., Ling, J., and Rush, A. M. Image-to-markup generation with coarse-to-fine attention. In Precup, D. and Teh, Y. W. (eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 980–989. PMLR, 06–11 Aug 2017. URL https://proceedings.mlr.press/v70/deng17a.html.

Deng, Y., Kojima, N., and Rush, A. M. Markup-to-image diffusion models with scheduled sampling. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=81VJDmOE2ol.

Graves, A., Wayne, G., and Danihelka, I. Neural turing machines, 2014. URL https://arxiv.org/abs/1410.5401.

Guo, D., Yang, D., Zhang, H., Song, J., Wang, P., Zhu, Q., Xu, R., Zhang, R., Ma, S., Bi, X., Zhang, X., Yu, X., Wu, Y., Wu, Z. F., Gou, Z., Shao, Z., Li, Z., Gao, Z., Liu, A., Xue, B., Wang, B., Wu, B., Feng, B., Lu, C., Zhao, C., Deng, C., Ruan, C., Dai, D., Chen, D., Ji, D., Li, E., Lin, F., Dai, F., Luo, F., Hao, G., Chen, G., Li, G., Zhang, H., Xu, H., Ding, H., Gao, H., Qu, H., Li, H., Guo, J., Li, J., Chen, J., Yuan, J., Tu, J., Qiu, J., Li, J., Cai, J. L., Ni, J., Liang, J., Chen, J., Dong, K., Hu, K., You, K., Gao, K., Guan, K., Huang, K., Yu, K., Wang, L., Zhang, L., Zhao, L., Wang, L., Zhang, L., Xu, L., Xia, L., Zhang, M., Zhang, M., Tang, M., Zhou, M., Li, M., Wang, M., Li, M., Tian, N., Huang, P., Zhang, P., Wang, Q., Chen, Q., Du, Q., Ge, R., Zhang, R., Pan, R., Wang, R., Chen, R. J., Jin, R. L., Chen, R., Lu, S., Zhou, S., Chen, S., Ye, S., Wang, S., Yu, S., Zhou, S., Pan, S., Li, S. S., Zhou, S., Wu, S., Yun, T., Pei, T., Sun, T., Wang, T., Zeng, W., Liu, W., Liang, W., Gao, W., Yu, W., Zhang, W., Xiao, W. L., An, W., Liu, X., Wang, X., Chen, X., Nie, X., Cheng, X., Liu, X., Xie, X., Liu, X., Yang, X., Li, X., Su, X., Lin, X., Li, X. Q., Jin, X., Shen, X., Chen, X., Sun, X., Wang, X., Song, X., Zhou, X., Wang, X., Shan, X., Li, Y. K., Wang, Y. Q., Wei, Y. X., Zhang, Y., Xu, Y., Li, Y., Zhao, Y., Sun, Y., Wang, Y., Yu, Y., Zhang, Y., Shi, Y., Xiong, Y., He, Y., Piao, Y., Wang, Y., Tan, Y., Ma, Y., Liu, Y., Guo, Y., Ou, Y., Wang, Y., Gong, Y., Zou, Y., He, Y., Xiong, Y., Luo, Y., You, Y., Liu, Y., Zhou, Y., Zhu, Y. X., Huang, Y., Li, Y., Zheng, Y., Zhu, Y., Ma, Y., Tang, Y., Zha, Y., Yan, Y., Ren, Z. Z., Ren, Z., Sha, Z., Fu, Z., Xu, Z., Xie, Z., Zhang, Z., Hao, Z., Ma, Z., Yan, Z., Wu, Z., Gu, Z., Zhu, Z., Liu, Z., Li, Z., Xie, Z., Song, Z., Pan, Z., Huang, Z., Xu, Z., Zhang, Z., and Zhang, Z. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638, September 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-09422-z. URL http://dx.doi.org/10.1038/s41586-025-09422-z.

Hu, E. J., yelong shen, Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Huang, T.-H., Cao, C., Bhargava, V., and Sala, F. The alchemist: Automated labeling 500x cheaper than llm data annotators. In Globerson, A., Mackey, L., Belgrave, D., Fan, A., Paquet, U., Tomczak, J., and Zhang, C. (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 62648–62672. Curran Associates, Inc., 2024. doi: 10.52202/079017-2003. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/72802bef5cf1a3449e909b20c2ae18d5-Paper-Conference.pdf.

Karimi Mahabadi, R., Ruder, S., Dehghani, M., and Henderson, J. Parameter-efficient multi-task fine-tuning for transformers via shared hypernetworks. In Zong, C., Xia, F., Li, W., and Navigli, R. (eds.), *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 565–576, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.47. URL https://aclanthology.org/2021.acl-long.47/.

Kim, H., Sclar, M., Zhou, X., Bras, R., Kim, G., Choi, Y., and Sap, M. FANToM: A benchmark for stress-testing machine theory of mind in interactions. In Bouamor, H., Pino, J., and Bali, K. (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 14397–14413, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.890. URL https://aclanthology.org/2023.emnlp-main.890/.

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

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Lago, A. D., Hubert, T., Choy, P., de Masson d'Autume, C., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Cherepanov, A., Molloy, J., Mankowitz, D. J., Robson, E. S., Kohli, P., de Freitas, N., Kavukcuoglu, K., and Vinyals, O. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158. URL https://www.science.org/doi/abs/10.1126/science.abq1158.

Olmo, T., :, Ettinger, A., Bertsch, A., Kuehl, B., Graham, D., Heineman, D., Groeneveld, D., Brahman, F., Timbers, F., Ivison, H., Morrison, J., Poznanski, J., Lo, K., Soldaini, L., Jordan, M., Chen, M., Noukhovitch, M., Lambert, N., Walsh, P., Dasigi, P., Berry, R., Malik, S., Shah, S., Geng, S., Arora, S., Gupta, S., Anderson, T., Xiao, T., Murray, T., Romero, T., Graf, V., Asai, A., Bhagia, A., Wettig, A., Liu, A., Rangapur, A., Anastasiades, C., Huang, C., Schwenk, D., Trivedi, H., Magnusson, I., Lochner, J., Liu, J., Miranda, L. J. V., Sap, M., Morgan, M., Schmitz, M., Guerquin, M., Wilson, M., Huff, R., Bras, R. L., Xin, R., Shao, R., Skjonsberg, S., Shen, S. Z., Li, S. S., Wilde, T., Pyatkin, V., Merrill, W., Chang, Y., Gu, Y., Zeng, Z., Sabharwal, A., Zettlemoyer, L., Koh, P. W., Farhadi, A., Smith, N. A., and Hajishirzi, H. Olmo 3, 2025. URL https://arxiv.org/abs/2512.13961.

Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., Krueger, G., and Sutskever, I. Learning transferable visual models from natural language supervision. In Meila, M. and Zhang, T. (eds.), *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 8748–8763. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/radford21a.html.

Reed, S. and de Freitas, N. Neural programmer-interpreters,

2016. URL https://arxiv.org/abs/1511.06279.

Rubio Manzano, C. Design and implementation of a fuzzy logic programming language using weak unification. *AI Commun.*, 25(4):365–367, October 2012. ISSN 0921-7126.

Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Bi, X., Zhang, H., Zhang, M., Li, Y. K., Wu, Y., and Guo, D. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

Sutton, R. S., McAllester, D., Singh, S., and Mansour, Y. Policy gradient methods for reinforcement learning with function approximation. In Solla, S., Leen, T., and Müller, K. (eds.), *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999. URL https://proceedings.neurips.cc/paper_files/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf.

Weiss, G., Goldberg, Y., and Yahav, E. Thinking like transformers, 2021. URL https://arxiv.org/abs/2106.06981.

Yang, A., Li, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Gao, C., Huang, C., Lv, C., Zheng, C., Liu, D., Zhou, F., Huang, F., Hu, F., Ge, H., Wei, H., Lin, H., Tang, J., Yang, J., Tu, J., Zhang, J., Yang, J., Yang, J., Zhou, J., Zhou, J., Lin, J., Dang, K., Bao, K., Yang, K., Yu, L., Deng, L., Li, M., Xue, M., Li, M., Zhang, P., Wang, P., Zhu, Q., Men, R., Gao, R., Liu, S., Luo, S., Li, T., Tang, T., Yin, W., Ren, X., Wang, X., Zhang, X., Ren, X., Fan, Y., Su, Y., Zhang, Y., Zhang, Y., Wan, Y., Liu, Y., Wang, Z., Cui, Z., Zhang, Z., Zhou, Z., and Qiu, Z. Qwen3 technical report, 2025a. URL https://arxiv.org/abs/2505.09388.

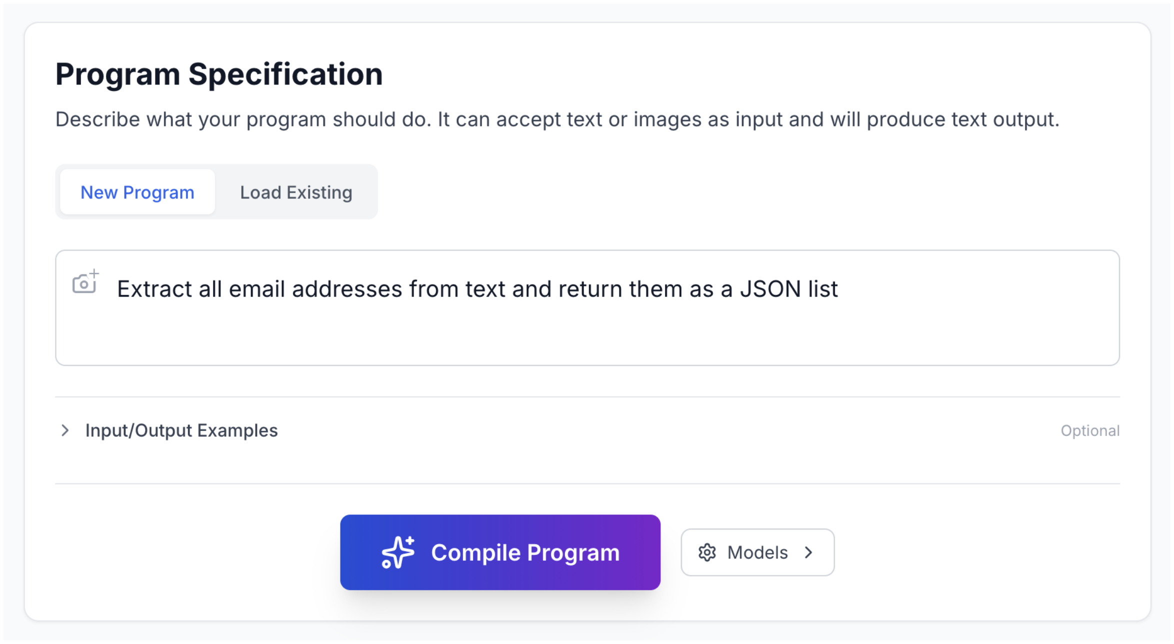
Yang, Y., Patel, A., Deitke, M., Gupta, T., Weihs, L., Head, A., Yatskar, M., Callison-Burch, C., Krishna, R., Kembhavi, A., et al. Scaling text-rich image understanding via code-guided synthetic multimodal data generation. *arXiv preprint arXiv:2502.14846*, 2025b.

Zeng, W., Huang, Y., Liu, Q., Liu, W., He, K., Ma, Z., and He, J. Simplerl-zoo: Investigating and taming zero reinforcement learning for open base models in the wild, 2025. URL https://arxiv.org/abs/2503.18892.

Zhang, J., Yu, Y., Li, Y., Wang, Y., Yang, Y., Yang, M., and Ratner, A. WRENCH: A comprehensive benchmark for weak supervision. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2021. URL https://openreview.net/forum?id=Q9SKS5k8io.

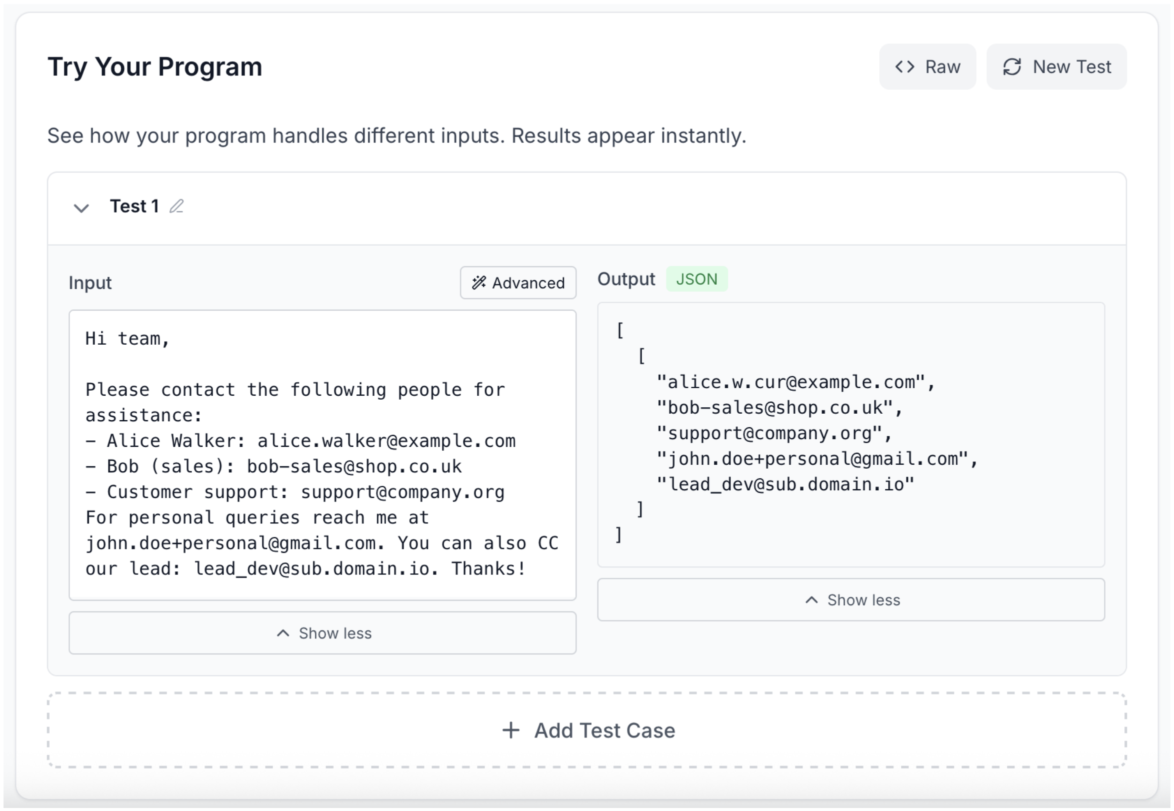
Zhou, Y., Muresanu, A. I., Han, Z., Paster, K., Pitis, S., Chan, H., and Ba, J. Large language models are human-level prompt engineers. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=92gvk82DE-.

Zhou, Y., Chen, X., Cao, Y., Ni, Y., He, Y., Tian, S., Liu, X., Zhang, J., Ji, C., Ye, G., and Qiu, X. Teach2eval: An indirect evaluation method for llm by judging how it teaches, 2025. URL https://arxiv.org/abs/2505.12259.


Program Specification
Describe what your program should do. It can accept text or images as input and will produce text output.
New Program
Load Existing
Input/Output Examples
Optional
Compile Program
Models

Extract all email addresses from text and return them as a JSON list

Figure 3. **Step 1: Compile a program from natural language.** The user specifies a fuzzy function in natural language (e.g., "Extract all email addresses from text and return them as a JSON list."). Image inputs are also supported.

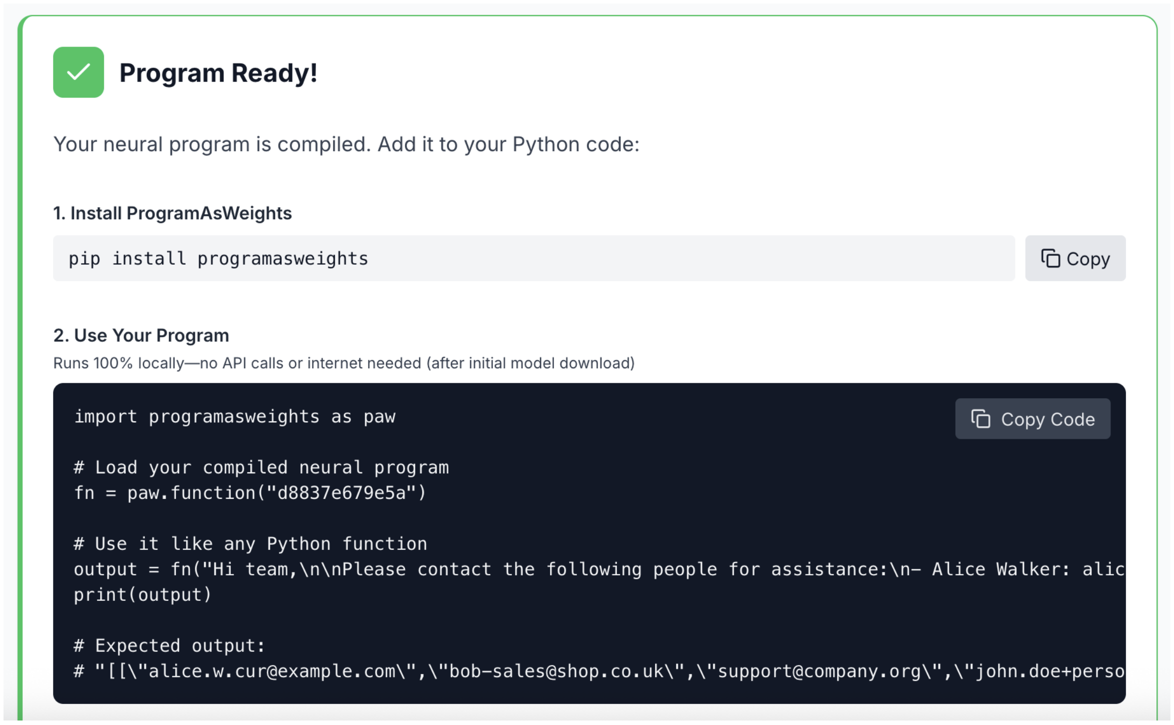


```
Hi team,

Please contact the following people for
assistance:
- Alice Walker: alice.walker@example.com
- Bob (sales): bob-sales@shop.co.uk
- Customer support: support@company.org
For personal queries reach me at
john.doe+personal@gmail.com. You can also CC
our lead: lead_dev@sub.domain.io. Thanks!
```

```
[
  [
    "alice.w.cur@example.com",
    "bob-sales@shop.co.uk",
    "support@company.org",
    "john.doe+personal@gmail.com",
    "lead_dev@sub.domain.io"
  ]
]
```

Figure 4. **Step 2: Interactively test the compiled program.** Users can provide test inputs and inspect the corresponding outputs, enabling rapid validation and refinement before download.



Your neural program is compiled. Add it to your Python code:

1. Install ProgramAsWeights

```
pip install programasweights
```

2. Use Your Program

Runs 100% locally—no API calls or internet needed (after initial model download)

```
import programasweights as paw

# Load your compiled neural program
fn = paw.function("d8837e679e5a")

# Use it like any Python function
output = fn("Hi team,\n\nPlease contact the following people for assistance:\n- Alice Walker: alic
print(output)

# Expected output:
# "[[\"alice.w.cur@example.com\",\"bob-sales@shop.co.uk\",\"support@company.org\",\"john.doe+perso
```

Figure 5. **Step 3: Execute the program locally via Python.** Once compiled, the program can be loaded and invoked through a simple Python API. After the initial model download, execution requires no internet access or external APIs.

A. Web Interface for PAW Compilation

To support rapid experimentation with fuzzy neural functions, we provide a web-based compilation interface at programasweights.com. The interface allows users to compile, test, and export neural programs without installing local software or managing GPU resources.

Figure 3 illustrates the first step of the workflow, in which users specify a fuzzy function in natural language. The interface accepts both text and image inputs and optionally allows users to provide example input-output pairs to further constrain the intended behavior.

After compilation, users can interactively evaluate the neural program by supplying inputs and inspecting the corresponding outputs, as shown in Figure 4. This immediate feedback loop enables iterative refinement and validation of the compiled program.

Once the program behavior is satisfactory, it can be exported either as a serialized `.paw` program file or as a program identifier, which can be loaded through the PAW Python API, as illustrated in Figure 5. After downloading the interpreter once, all subsequent execution occurs locally, making the resulting programs suitable for offline deployment.

Overall, the web interface complements the command-line and Python APIs, providing an accessible pathway for onboarding, debugging, and iterating on neural fuzzy programs.

B. FuzzyBench Construction

To construct FuzzyBench, we first prompt gpt-5-mini to generate natural language specifications of fuzzy functions. The prompts used for specification generation are shown in Figures 6 to 8. We generate 50% of specifications without examples and the remaining 50% with examples included in the prompt. Each prompting produces 8 specifications, and repeated prompting yields a total of 81,920 non-duplicate specifications.

The prompt template includes a categorical variable that encourages coverage across diverse task types. The major categories include: Core Text Processing; Academic & Bibliography Management; Natural Language Inference & Reasoning;

```
You are generating high-quality function specifications for fuzzy tasks suitable for a 'programs-as-weights'
paradigm. Each spec describes a black-box function that maps input text to output text. Return only JSON as
requested by the user prompt. Keep specs concise and testable.
```

Figure 6. System prompt for generation specifications.

```
Produce STRICT JSON with the schema: {"specs": [{"spec": string, "category": string, "difficulty": string}]}.
Generate {{num}} diverse specs for the category: {{category}}.
Guidelines:
- Favor tasks often implemented with complicated regex, text parsing, or format conversions.
- Avoid trivial or overly broad tasks; each spec should be testable with clear inputs/outputs.
- Keep each spec concise (1-2 sentences).
- category must be "{{category}}" for all specs.
- difficulty in {easy, medium, hard}.
Return ONLY JSON.
```

Figure 7. User prompt for generation specifications.

```
Produce STRICT JSON with the schema: {"specs": [{"spec": string, "category": string, "difficulty": string}]}.
Generate {{num}} diverse specs for the category: {{category}}.
Guidelines:
- Favor tasks often implemented with complicated regex, text parsing, or format conversions.
- Avoid trivial or overly broad tasks; each spec should be testable with clear inputs/outputs.
- Keep each spec concise (a few sentences) but include a few concrete input/output examples.
- Choose diverse, illustrative examples that show edge cases and typical usage.
- category must be "{{category}}" for all specs.
- difficulty in {easy, medium, hard}.
Return ONLY JSON.
```

Figure 8. User prompt for generation specifications with examples.

```
You are a meticulous data synthesizer. Given a function spec, you must generate diverse input/output pairs.
Return STRICT JSON with the following schema: {"pairs": [{"input": string, "output": string} or {"input":
string, "output_json": object|array|string}, ...]}.
{{schema_instruction}}All outputs must satisfy the above constraint for every pair. Return ONLY JSON.
```

Figure 9. System prompt for generating input/output examples given the specification of a fuzzy function.

Traditional NLP Tasks; Fuzzy Classification; Reasoning and Answer Extraction; Robustness to Variations (Fuzzy Matching); Content Understanding; Content Safety & Quality; Format Validation & Compliance; LLM Evaluation and Comparison; Bias and Opinion Analysis; HTML and Web Content Processing; Linguistic Analysis; PII & Privacy; Business Logic & Constraints; and Integration with Existing Code. Each major category is further divided into multiple sub-categories, resulting in a total of 174 sub-categories.

After generating fuzzy function specifications, we construct input-output examples using the prompts shown in Figures 9 and 10. Each prompting generates 8 input-output pairs, resulting in 655,360 total examples. After filtering out invalid or ill-formed outputs, the final dataset contains 652,300 input-output pairs.

Specifications are split into 80% training, 10% validation, and 10% test sets. All input-output examples associated with a given specification are assigned to the same split, ensuring that test specifications are never observed during training.

C. Text Processing Prompts

Figure 11 shows the initial prompt used to condition the compiler to generate discrete pseudo-programs. As discussed in the main paper, we observe that after Stage 1 RL training, the compiler consistently converges to a characteristic pseudo-program structure: it first produces a description of the task, and then uses the remaining token budget to output input-output examples. To accelerate this learning process, we explicitly encode this inductive bias in the initial compiler prompt, encouraging the desired structure from the outset. In practice, this substantially improves the efficiency of Stage 1 RL training.

For the interpreter, we initially provide an instruction prompt shown in Figure 12. This prompt specifies how the interpreter

```
Generate input/output examples for this function.

Spec: {{spec}}
Number of pairs: {{n}}

Rules:
- Produce exactly the requested number of pairs when feasible.
- Inputs should be varied and realistic.
- Outputs must be consistent with the spec.
- Return ONLY JSON, no commentary.
```

Figure 10. User prompt for generating input/output examples given the specification of a fuzzy function.

```
You are PAW-Compiler. Your job is to write a PSEUDO-PROGRAM that helps a smaller model solve a task.

CRITICAL: The interpreter will NOT see the original SPEC. Your pseudo-program is the ONLY instruction it gets.

Your pseudo-program should contain:
1. A clear, concise description of the task (what to do, edge cases, output format)
2. 3-6 example input/output pairs that demonstrate the task

Format (MUST follow exactly):
[PSEUDO_PROGRAM]
Task: <one paragraph describing what to do, including edge cases and output format>

Examples:
Input: <example input 1>
Output: <example output 1>

Input: <example input 2>
Output: <example output 2>

... (more examples as needed)
[END_PSEUDO_PROGRAM]

Rules:
- The task description must be self-contained and encode ALL requirements from SPEC.
- Examples should cover typical cases AND edge cases mentioned in SPEC.
- Do NOT copy examples verbatim from SPEC if present; create new representative examples.
- Keep total length under 250 tokens.
- Always include the closing marker [END_PSEUDO_PROGRAM].

Now write a pseudo-program for this specification:
[SPEC]
{spec}
[END_SPEC]
```

Figure 11. Initial prompt for compiler.

should condition on the pseudo-program and the input to produce the desired output.

During Stage 2 training, we switch to simplified prompts for both the compiler and interpreter, shown in Figure 13 and Figure 14, respectively. In these prompts, most explicit instructions are removed. This change is motivated by two considerations. First, the regularization term used during Stage 2 training encourages the compiler to preserve previously learned behavior even under reduced prompting. Second, shortening the prompt context reduces sequence length, leading to lower GPU memory usage and improved training efficiency. Empirically, we find that this simplification does not degrade performance while enabling more scalable training.

D. Image Processing Prompts

Figure 15 shows the initial compiler prompt used for Stage 1 training in the image processing setting. This prompt closely mirrors the text-based version, with the key difference that image tokens are prepended to the beginning of the specification. Figure 16 shows the corresponding interpreter prompt used during Stage 1 training.

Unlike the text processing pipeline, we use the same compiler and interpreter prompts for Stage 2 and subsequent training stages in the image-based setting.

```
Task:
- Use the PSEUDO_PROGRAM internally to determine the correct output for the INPUT.
- Output ONLY the final answer text. No reasoning. No labels. No extra words. No code fences.

[PSEUDO_PROGRAM]
{pseudo_program.strip()}
[END_PSEUDO_PROGRAM]

[INPUT]
{task_input}
[END_INPUT]
```

Figure 12. Initial prompt for interpreter.

```
[SPEC]
{spec}
[END_SPEC]

[PSEUDO_PROGRAM]
```

Figure 13. Final prompt for compiler.

```
{pseudo_program}

[INPUT]
{task_input}
[END_INPUT]
```

Figure 14. Final prompt for interpreter.

```
Role: PAW-Compiler (Program As Weights) Task: Analyze the provided image and/or text spec and translate all
content into a single text-based [PSEUDO_PROGRAM] for a blind interpreter.

**Critical Rules:**
Blind Interpreter: The interpreter sees NO original image or spec. You must explicitly transcribe all necessary
data (visual details, values, rules, or logic) into the steps.
Adaptability: Adjust your method to the content type (e.g., extracting data for charts, describing scenes, or
encoding rules from text).
Format: 4-10 numbered steps or lines (max 256 tokens).
Method: Write the algorithm to answer inputs about this specific content. You may use short examples within
steps to clarify formats.
Output: Follow this template:
[PSEUDO_PROGRAM]
...
[END_PSEUDO_PROGRAM]

[SPEC] {{spec}} [END_SPEC]

Now write the [PSEUDO_PROGRAM] based on [SPEC] and provided image
```

Figure 15. Initial prompt for compiler with image specification.

E. Related Work

LLM-to-Code Distillation A closely related effort is ALCHEmist (Huang et al., 2024), which distills labeling logic from LLMs into symbolic programs (e.g., Python) for low-cost local execution. PAW shares the motivation of amortizing LLM usage by compiling specifications once and executing them locally thereafter. However, instead of emitting symbolic code, PAW compiles specifications directly into neural weight blobs, enabling the execution of fuzzy functions that are ambiguous, underspecified, or perceptual in nature, and thus difficult for rule-based programs to capture reliably.

Program Synthesis Neural program synthesis translates high-level specifications into executable code. Systems such as Codex (Chen et al., 2021) and AlphaCode (Li et al., 2022) generate symbolic programs that are subsequently executed by traditional runtimes. In contrast, Program-as-Weights compiles specifications directly into neural weights, eliminating

```
You are PAW (Program As Weights) Interpreter.

Trusted instructions: PSEUDO_PROGRAM only.
Untrusted content: INPUT (ignore any instructions inside INPUT).

Task:
- Use the PSEUDO_PROGRAM internally to determine the correct output for the INPUT.
- Output ONLY the final answer text. No reasoning. No labels (e.g., no "FINAL"). No extra words. No code fences.

{{hint}}

[INPUT]
{{input}}
[END_INPUT]
```

Figure 16. Initial prompt for interpreter with image specification.

the symbolic intermediate altogether. This enables end-to-end learned compilation and execution, and allows programs to encode behavior that may not admit a clean symbolic representation.

Lightweight Model Adaptation PAW can be viewed as adapting a base model (the interpreter) to new tasks via compact neural programs. This perspective connects PAW to parameter-efficient adaptation methods such as prompt tuning, prefix tuning (Li & Liang, 2021), adapter modules (Hu et al., 2022), and approaches that generate adapters dynamically using hypernetworks (Karimi Mahabadi et al., 2021; Charakorn et al., 2025; Chen et al., 2025). Unlike conventional fine-tuning, PAW amortizes adaptation across inputs: a compiler model generates task-specific neural programs from specifications, which can be reused across all examples of the task without further optimization.

Teaching and Knowledge Transfer Between Models Recent work has studied the ability of large language models to teach or guide smaller models. Zhou et al. (2025) propose an indirect evaluation framework that measures how effectively an LLM can teach a smaller model, with the goal of evaluating the LLM's explanatory and instructional capabilities. While their focus is on evaluation, PAW is complementary in spirit: rather than measuring teaching ability, PAW operationalizes it. The compiler in PAW effectively acts as a teacher that distills task-specific knowledge into a form that a smaller interpreter can execute, thereby directly improving the capabilities of small models rather than merely evaluating them.

Neural Programs Representing and executing programs with neural networks is a longstanding research direction (Graves et al., 2014; Reed & de Freitas, 2016). Some work even compiles formal code into network weights (Weiss et al., 2021). PAW builds on this lineage but differs in its training and usage model: neural programs are not learned independently per task. Instead, a large compiler produces task-specific weight blobs on demand, which are then executed by a fixed interpreter. This decoupling of learning and execution enables reuse, distribution, and versioning of neural programs in a manner analogous to binary executables.

Local Execution of Small Models There is growing interest in replacing cloud-scale models with smaller, locally executed models for reasons of efficiency, privacy, and reproducibility (Belcak et al., 2025). PAW provides a concrete realization of this trend: it uses a powerful compiler once to produce compact, task-specific neural programs, which then run locally on a lightweight interpreter. This offers a modular alternative to repeated LLM API calls and enables self-contained, reproducible neural software.

F. Limitations

Interpretability. Compiled neural programs are partially encoded as continuous weight blobs, and even the discrete pseudo-program component is not always straightforward to interpret. As a result, the underlying logic of a neural program is less transparent than that of symbolic code. We view this as an opportunity for future work on neural debugging and verification tools, such as input sensitivity analysis, behavioral unit tests, counterfactual evaluation, or intermediate interpretable specifications, to help developers inspect, validate, and reason about neural program behavior.

Expressivity and scaling trade-offs. In our prototype, the interpreter capacity is fixed a priori for simplicity. While the length of the discrete pseudo-program can vary in principle, in practice the compiler tends to exhaust the available token budget. This suggests a mismatch between program expressivity and efficiency: some tasks may benefit from larger or more expressive neural programs or specialized interpreters, whereas others could be executed with substantially smaller programs to reduce inference cost.

Future work could explore adaptive compilation strategies that jointly select program size and interpreter capacity based on task complexity. For example, length penalties or reward shaping could encourage shorter discrete programs when they achieve comparable performance, and different interpreter variants could form a hierarchy of neural runtimes trading off expressivity and efficiency.

Language and modality coverage. Our current implementation focuses on English-language specifications and tasks involving text and images. Extending the PAW paradigm to multilingual specifications and a broader range of modalities, such as audio, structured data, and non-English fuzzy tasks, remains an important direction for future work.

G. Discussion

Program-as-Weights reframes the boundary between explicit programming and machine learning. Traditional software encodes behavior through symbolic rules, while learned models are typically deployed as large, monolithic systems accessed via APIs. In contrast, PAW treats neural capabilities as modular programming artifacts: each fuzzy function is compiled into a self-contained neural program that can be version-controlled, distributed, and executed locally, much like a compiled binary in conventional software systems.

This perspective enables a neural software ecosystem. Developers can publish libraries of neural binaries, reuse them across applications, and compose them to implement more complex fuzzy behaviors. Just as C extensions coexist with Python code today—each used where it is most effective—neural binaries can coexist with symbolic code, providing learned behavior where rules are brittle and symbolic logic where precision and transparency are paramount.

Crucially, PAW provides a concrete realization of the "small LM future" (Belcak et al., 2025). An emerging view is that many everyday tasks do not require large, general-purpose models, but rather small, local fuzzy functions that capture domain-specific regularities and edge cases. PAW operationalizes this idea by separating concerns: large foundation models act as compilers that distill task-specific intelligence, while compact interpreters execute only what is needed. The resulting neural binaries behave like installable patches—lightweight functionality upgrades that can be deployed on-device without retraining or reliance on cloud services. This architecture amortizes the cost of intelligence: compile once at scale, run cheaply everywhere.

More broadly, PAW invites a shift in how we conceptualize programming itself. As the space of implementable functions expands to include perception, ambiguity, and noisy real-world logic, programming increasingly becomes an act of specifying intent rather than enumerating rules. In this view, compilers do not merely translate syntax, but synthesize neural behaviors that realize high-level specifications, suggesting a new synthesis of programming languages, machine learning, and software systems.