SELF-POLICY DISTILLATION VIA CAPABILITY-SELECTIVE SUBSPACE PROJECTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Self-distillation bootstraps large language models (LLMs) by training on their own generations. However, existing methods either rely on external signals to curate self-generated outputs (e.g., correctness filtering, execution feedback, and reward search), which are costly and unavailable for the best-performing frontier models, or skip curation entirely and train on all raw outputs, an approach that is often domain-specific and hard to generalize. Both also share a deeper weakness that self-generated outputs entangle task-relevant capability with others, such as stylistic patterns, formatting artifacts, and model-specific errors, diluting the signal for the specific capability one aims to improve. In this paper, we propose **Self-Policy Distillation (SPD)**, which achieves generalizable, capability selective with only minimal external signal ($< 5\%$ labeled data in each domain). Specifically, SPD extracts a low-rank capability subspace from the model's own gradients on correctness-defining tokens, projects key-value (KV) activations into this subspace during self-generation, and fine-tunes on the resulting raw outputs with standard next-token prediction loss. Through extensive experiments across code generation, mathematical reasoning, and multiple-choice QA, we show that SPD achieves up to 13% improvement over state-of-the-art self-distillation methods and up to 16.3% improvement over pre-trained baselines. Notably, SPD demonstrates superior generalizability, achieving 15.7% better performance under out-of-domain generalization settings.



Figure 1: SPD comparisons and performance summary. Top left: comparison with existing self-distillation methods across three key axes. Top right: average improvement of SPD over the base model for each of the five LLM backbones, computed across three capability domains and six datasets. Bottom: SPD improves base model performance across domain-specific benchmarks, in-domain transfer, and out-domain transfer settings. All metrics in the figure are reported as accuracy (%).

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.

1 INTRODUCTION

Self-distillation has emerged as a powerful paradigm for improving large language models (LLMs) by training them on their own generations [26, 33, 35]. Existing methods typically do not use raw self-generated outputs directly, because doing so can reinforce the model's existing errors and stylistic artifact, leading to results in poor performance [8, 9, 20, 21]. Instead, one line of works relies on external signals to curate or guide self-generated data, such as correctness filtering [33], verifier-based selection [16], consistency-driven rationale evaluation [19], or reward-guided search [34]. While effective, these signals introduce additional cost and task-specific infrastructure. In addition, they may be unavailable for the strongest frontier models, where more reliable labels, verifiers, or interactive feedback are often hard to obtain [23, 27]. Another line of work skips explicit curation entirely and trains directly on raw self-generated outputs after internal filtering. Simple self-distillation (SSD) [35] shows that this simple strategy can improve model performance without using a verifier, reward model, or reinforcement learning (RL). However, SSD is designed for code generation, and its generalization beyond the code domain remains unclear.

Additionally, both lines of work share a deeper weakness: self-generated outputs are a mixed source of supervision. They contain not only the task-relevant capability signal we aim to preserve, but also model-specific errors and noisy artifacts that can hinder capability transfer [9, 20], as well as stylistic patterns, rationale formats, and other non-essential reasoning details that may dilute the signal for the specific capability being improved [8, 21]. figure 1 (top-left) summarizes that existing self-distillation paradigms each fall short on two of these key axes. These bottlenecks raise a central question:

*Can self-distillation be made with only minimal external signals by **internally steering** self-generated outputs to **generalize** across domains?*

To address this question, we propose **Self-Policy Distillation (SPD)**, a generalizable and capability-selective framework that requires only minimal external signals. SPD consists of two phases. In the first phase, SPD uses a small calibration set, accounting for less than 5% of the training data in each domain, to extract a low-rank capability subspace from the model's own gradients. These gradients are computed using a loss defined only on *correctness-defining* token positions. In practice, for each calibration example, we compute a loss only over the task-relevant answer span. We then collect the gradients with respect to the key and value activations at selected layers. Finally, we perform singular value decomposition (SVD) on these gradients to obtain a low-rank projection subspace for the key (K) and value (V) activations, which are used to construct the *projection hooks* in second phase.

In the second phase, we use this capability subspace to steer self-generation. Specifically, we register projection hooks on the corresponding KV activations during decoding. These hooks do not modify the model parameters. Instead, they filter intermediate representations so that the model's self-generated outputs are biased toward the target capability. Using the hooked model, we generate one raw completion for each training prompt and construct a self-generated corpus for distillation. After data generation, all hooks are removed, and the original model is fine-tuned on the resulting prompt–completion pairs with the standard loss. Therefore, SPD remains a fully self-contained self-distillation pipeline that requires only minimal external signals, while the projection hooks help selectively preserve target capability signals and filter out task-irrelevant noise.

We evaluate SPD across three capability domains — code generation, mathematical reasoning, and multiple-choice question-answering (QA) — under two transfer settings that test whether learned gains generalize within a capability domain (**in-domain**) and beyond it (**out-of-domain**). As summarized in figure 1, SPD consistently improves over the base model across five backbones (top-right), three source benchmarks (bottom-left), and both transfer settings (bottom-middle and bottom-right). Specifically, SPD achieves up to 13% improvement over state-of-the-art self-distillation approaches and up to 16.3% improvement over pre-trained baselines under the in-domain setting. Notably, SPD also demonstrates strong generalization, achieving up to 15.7% better performance under the out-of-domain setting.

To summarize, our contributions are outlined below:

- We propose **Self-Policy Distillation (SPD)**, a new self-distillation paradigm that trains on selective self-generated outputs. SPD requires only minimal external signals and generalizes across domains by selectively preserving target capability signals during self-generation.
- We define *correctness spans*, which are token positions that directly determine task success, and apply a correctness-aligned loss only on these positions. SVD on the resulting K/V activation gradients yields a low-rank capability subspace that steers self-generation toward the target capability.

- Across code generation, mathematical reasoning, and multiple-choice QA, SPD consistently outperforms state-of-the-art self-distillation methods under both in-domain and out-of-domain generalization settings.

2 Related Work

Distillation. Early off-policy distillation methods mainly focus on transferring information from a stronger teacher to a student model, either through softened output distributions [15] or sequence-level supervision [18]. However, such off-policy training introduces a train-inference distribution mismatch that leads to compounding errors at test time [7]. Later, on-policy distillation (OPD) [1, 4] mitigates this mismatch by using student rollouts while receiving per-token supervision from the teacher. However, it requires repeated student rollouts and heavy teacher supervision, making it costly and hard to scale. Self-distillation mitigates these limitations by replacing the external teacher with supervision derived from the model itself [2, 34, 35]. However, directly using self-generated outputs for training can reinforce the model's existing errors and stylistic artifact, leading to poor performance [8, 9, 20, 21]. Therefore, improving the quality of self-generated data is critical. Existing methods rely on external signals to curate self-generated data, such as correctness filtering [33], consistency or execution-based feedback [11, 19], or reward-guided search [34]. While effective, these signals introduce additional cost,task-specific infrastructure and unavailable for the best-performing frontier models. More recently, SSD [35] shows that a LLM can improve by training directly on its own raw outputs, without a verifier, a teacher model, or RL. However, SSD is designed for code generation and trained on domain-specific synthetic code data, leaving its generalization beyond the code domain unclear. In contrast, our work keeps the self-distillation setting with only minimal external signals, while making self-generated data capability-selective and generalizable across domains. A more detailed discussion comparing SPD (ours) with existing methods is in Appendix C.

Representation-level control. Recent work suggests that meaningful reasoning behavior and task-relevant information can often be compressed, manipulated, or steered in latent or KV-space representations rather than only at the output level [5, 12, 25, 30]. In parallel, prior work on distillation has shown that intermediate representations can provide richer and more targeted supervision than output logits alone [24, 28]. Our method combines these two perspectives in a teacher-free self-distillation setting. Specifically, rather than aligning a student to an external teacher's hidden states, we extract a capability-selective subspace from the student's own gradients and use it to steer self-generation, producing selective raw data for more useful self-supervision.

3 Method

In this section, we illustrate **self-policy distillation (SPD)**, a capability-selective framework that requires only minimal external signals. On a high level, SPD operates in two phases: Phase 1 (section 3.2) extracts low-rank K/V capability subspaces from gradients computed on a small calibration set using correctness-aligned loss, and Phase 2 (section 3.3) uses these subspaces as projection hooks to steer self-generation without modifying model parameters. The hooked model produces raw completions, after which the hooks are removed and the original model is fine-tuned on the resulting prompt–completion pairs using standard next-token prediction loss.

3.1 Preliminaries

We denote the calibration set by $\mathcal{D}_{\text{cal}} = \{(q^{(i)}, y^{(i)})\}_{i=1}^{N_{\text{cal}}}$, where each example $(q^{(i)}, y^{(i)})$ contains both a prompt and its answer. We further denote the training-prompt set by $\mathcal{D}_{\text{train}} = \{q^{(i)}\}_{i=1}^{N_{\text{train}}}$, where each example contains only the prompt. After self-generation, we obtain a self-generated corpus $\mathcal{D}_{\text{corpus}} = \{(q^{(i)}, \hat{y}^{(i)})\}_{i=1}^{N_{\text{train}}}$, where $\hat{y}^{(i)}$ is the model-generated output for prompt $q^{(i)}$. Finally, let f_θ denote a LLM with L layers, and let $\mathcal{L} \subseteq \{1, \ldots, L\}$ denote the set of target layers on which our method operates. Motivated by prior work showing that intermediate representations provide useful training signals [24, 28], we by default apply projection to both K and V at a middle and a late layer, ($\mathcal{L} = L, \lfloor L/2 \rfloor$), unless otherwise noted.

Distillation paradigm comparison. Let f_T denote an external teacher model, $f_{\theta_{\text{old}}}$ denote the student model before the distillation update, and f_{off} denote an offline model induced by a teacher, or a fixed offline dataset. Let p denote the corresponding next-token probability distribution, and let $z = T(q, y) = (z_1, \ldots, z_T)$ be tokenized sequence. Many distillation methods can be written as

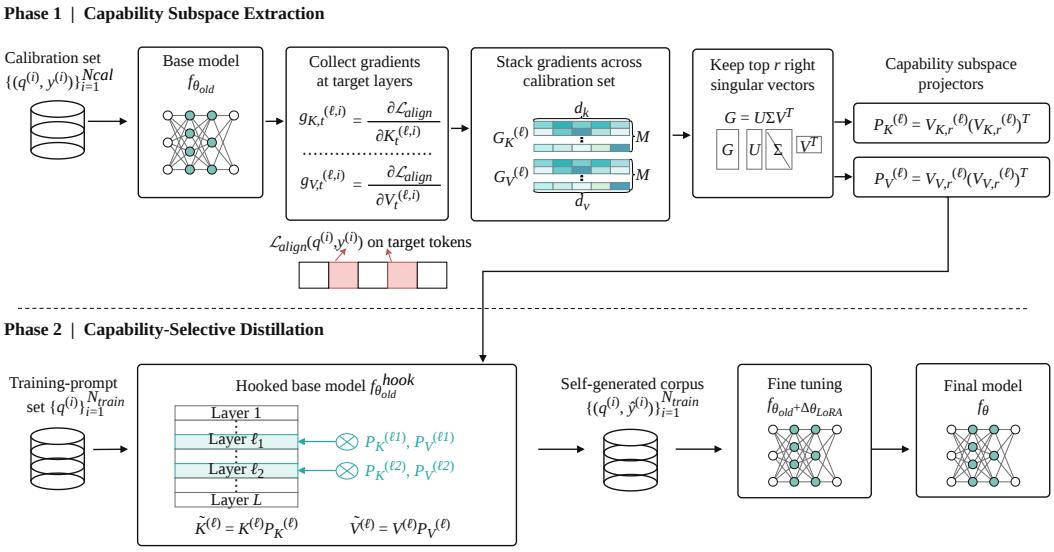

Figure 2: Overview of SPD. SPD operates in two phases: Phase 1 (top) (section 3.2) extracts low-rank K/V capability subspaces from gradients computed on a small calibration set using correctness-aligned loss, and Phase 2 (bottom) (section 3.3) uses these subspaces as projection hooks to steer self-generation without modifying model parameters. The hooked model produces raw completions, after which the hooks are removed and the original model is fine-tuned on the resulting prompt–completion pairs using standard next-token prediction loss.

matching a teacher distribution along trajectories sampled from a rollout policy f_{roll}:

$$\min_{\theta} \mathbb{E}_{q\sim\mathcal{D},\, y\sim f_{\text{roll}}(\cdot|q)} \sum_{t=1}^{T} \mathrm{KL}\left(p_T(\cdot \mid z_{<t}) \,\|\, p_\theta(\cdot \mid z_{<t})\right) \qquad z = T(q, y). \tag{1}$$

In off-policy distillation, the prefixes $z_{<t}$ come from an offline source, i.e., $f_{\text{roll}} = f_{\text{off}}$, where $f_{\text{off}} \in \{f_T, \mathcal{D}_{\text{offline}}\}$. In on-policy distillation, the prefixes are sampled from the student policy itself, e.g., $f_{\text{roll}} = f_{\theta_{\text{old}}}$. Both paradigms require an external teacher f_T to provide token-level supervision. Self-distillation, on the other hand, removes the external teacher and trains the model on its own generated outputs. To improve data quality, existing methods introduce an external scoring or filtering signal $S(q, y)$, leading to a weighted objective:

$$\min_{\theta} -\mathbb{E}_{q\sim\mathcal{D},\, y\sim f_{\theta_{\text{old}}}(\cdot|q)} \left[S(q, y) \sum_{t} \log p_\theta\left(z_t \mid z_{<t}\right) \right] \qquad z = T(q, y). \tag{2}$$

Thus, although the outputs are self-generated, the learning signal still largely depends on an external verifier, reward, or filtering mechanism.

Self-policy distillation (SPD). In contrast, SPD does not train directly on raw outputs from the original $f_{\theta_{\text{old}}}$. Instead, it first induces a self-policy through an internal transformation:

$$f_{\theta_{\text{old}}}^{\text{hook}} = \mathcal{T}_{\theta_{\text{old}}}^{\text{hook}}\left(f_{\theta_{\text{old}}}\right), \tag{3}$$

where $\mathcal{T}_{\theta_{\text{old}}}^{\text{hook}}$ is implemented by projection hooks derived from a capability-specific subspace. We then sample training outputs from this internally transformed policy:

$$\hat{y} \sim f_{\theta_{\text{old}}}^{\text{hook}}(\cdot \mid q). \tag{4}$$

Finally, the original model is trained to absorb this capability-steered behavior:

$$\min_{\theta} -\mathbb{E}_{q\sim\mathcal{D},\, \hat{y}\sim f_{\theta_{\text{old}}}^{\text{hook}}(\cdot|q)} \left[\sum_{t} \log p_\theta\left(z_t \mid z_{<t}\right) \right] \qquad z = T(q, \hat{y}). \tag{5}$$

Equivalently, we have $f_{\theta_{\text{old}}} \xrightarrow{\mathcal{T}^{\text{hook}}_{\theta_{\text{old}}}} f^{\text{hook}}_{\theta_{\text{old}}} \xrightarrow{\text{generate}} \hat{y} \xrightarrow{\text{distill}} f_\theta$. Therefore, SPD can be viewed as distilling an internally selected version of the model's own policy back into the original model. In the following sections, we describe the main procedure for extracting this capability-specific subspace and using it to induce the hooked self-policy. Algorithm 1 provides a detailed overview of the full procedure.

3.2 Capability Subspace Extraction

The goal of this phase is to select a low-dimensional subspace in the KV activation space that is most relevant to the target capability. Instead of using the standard next-token loss over the entire token sequence, we define a *correctness-aligned loss* that only select token positions directly tied to task success. This design helps the extracted subspace focus on capability-relevant directions, while reducing the influence of stylistic tokens, formatting artifacts, and other task-irrelevant noise. We then collect token-level gradients from each calibration example and concatenate them across the full calibration set to form gradient matrices for K and V. By applying SVD to these matrices, we retain the dominant directions in the KV dimensions and construct fixed projection matrices for later hook-based self-generation.

Correctness-aligned gradient collection. After tokenizing each calibration example $(q^{(i)}, y^{(i)})$, we denote the resulting token sequence by $T(q^{(i)}, y^{(i)}) = z^{(i)} = (z_1^{(i)}, \ldots, z_{T_i}^{(i)})$, where T_i is the token sequence length. Let $\mathcal{S}^{(i)} \subseteq \{1, \ldots, T_i\}$ denote the set of correctness-defining target token positions. These are token positions whose prediction is directly tied to task success. The details of how we select these token positions are provided in Appendix A.1. Formally, we define the correctness-aligned loss as

$$\mathcal{L}_{\text{align}}(q^{(i)}, y^{(i)}) = -\frac{1}{|\mathcal{S}^{(i)}|} \sum_{t \in \mathcal{S}^{(i)}} \log p_{\theta_{\text{old}}}\left(z_t^{(i)} \mid z_{<t}^{(i)}\right). \tag{6}$$

For positions $t \notin \mathcal{S}^{(i)}$, the labels are masked and do not contribute to the loss. Thus, the gradient signal is concentrated on tokens that are directly related to correctness.

For each calibration example $(q^{(i)}, y^{(i)})$, we perform a single forward–backward pass on the frozen student model. At each target layer $\ell \in \mathcal{L}$, we compute the gradient of the aligned loss with respect to the key and value activation vectors at every token position:

$$g_{K,t}^{(\ell,i)} = \frac{\partial \mathcal{L}_{\text{align}}(q^{(i)}, y^{(i)})}{\partial K_t^{(\ell,i)}} \in \mathbb{R}^{d_k}, \qquad g_{V,t}^{(\ell,i)} = \frac{\partial \mathcal{L}_{\text{align}}(q^{(i)}, y^{(i)})}{\partial V_t^{(\ell,i)}} \in \mathbb{R}^{d_v}, \tag{7}$$

where $t = 1, \ldots, T_i$; $g_{K,t}^{(\ell,i)}$, $g_{V,t}^{(\ell,i)}$ denote the key and value gradient vectors of token t at layer ℓ; d_k and d_v are the KV dimensions. Although gradients are defined for all token positions, only positions connected to the correctness-aligned loss receive task-relevant gradient signals.

We then collect these token-level gradients from each calibration example and concatenate them across the full calibration set to form gradient matrices for K and V:

$$G_K^{(\ell)} = \text{Concat}_{i,t}\left(g_{K,t}^{(\ell,i)}\right) \in \mathbb{R}^{M \times d_k}, \qquad G_V^{(\ell)} = \text{Concat}_{i,t}\left(g_{V,t}^{(\ell,i)}\right) \in \mathbb{R}^{M \times d_v}, \tag{8}$$

where the concatenation is taken over the token positions $(i,t) \in \{(i,t) : i = 1, \ldots, N_{\text{cal}},\ t = 1, \ldots, T_i\}$, and $M = \sum_{i=1}^{N_{\text{cal}}} T_i$. Therefore, each row of $G_K^{(\ell)}$ or $G_V^{(\ell)}$ corresponds to one token-level gradient direction in the KV feature space.

SVD and projection matrix. For each target layer $\ell \in \mathcal{L}$, we apply SVD to the collected gradient matrices. For the key (K) activations, we write

$$G_K^{(\ell)} = U_K^{(\ell)} \Sigma_K^{(\ell)} \left(V_K^{(\ell)}\right)^\top. \tag{9}$$

The right-singular vectors span directions in the K feature dimension. We retain the top-r right-singular vectors, $V_{K,r}^{(\ell)} \in \mathbb{R}^{d_k \times r}$, which correspond to the dominant gradient directions associated with correctness-defining tokens. The rank-r orthogonal projection matrix is then defined as

$$P_K^{(\ell)} = V_{K,r}^{(\ell)} \left(V_{K,r}^{(\ell)}\right)^\top \in \mathbb{R}^{d_k \times d_k}. \tag{10}$$

Similarly, for the value activations, we obtain

$$P_V^{(\ell)} = V_{V,r}^{(\ell)} \big(V_{V,r}^{(\ell)}\big)^\top \in \mathbb{R}^{d_v \times d_v}. \tag{11}$$

Intuitively, the top singular directions capture the KV dimensions most sensitive to the correctness-aligned loss. Directions associated with smaller singular values are treated as lower-energy variation and are less likely to represent stable capability-relevant information. Therefore, the projection matrices $P_K^{(\ell)}$ and $P_V^{(\ell)}$ preserve the selected capability subspace while suppressing noisier directions. These matrices are computed once per run and kept fixed during subsequent self-generation.

3.3 Capability-Selective Distillation

After obtaining the projection matrices $P_K^{(\ell)}$ and $P_V^{(\ell)}$ from capability subspace extraction (section 3.2), we directly apply them as *projection hooks* during self-generation. Specifically, at each target layer, the hook projects the KV activations onto the selected capability subspace. The goal is to produce self-generated outputs that are biased toward the target capability while keeping the model parameters unchanged. Finally, we fine-tune the original model on the resulting self-generated corpus.

Projection hooks. During autoregressive generation, we insert projection hooks at each target layer $\ell \in \mathcal{L}$. Specifically, Projection hooks are applied to the K and V activations:

$$\widetilde{K}^{(\ell)} = K^{(\ell)} P_K^{(\ell)}, \qquad \widetilde{V}^{(\ell)} = V^{(\ell)} P_V^{(\ell)}. \tag{12}$$

The subsequent attention computation uses $\widetilde{K}^{(\ell)}$ and $\widetilde{V}^{(\ell)}$ instead of the original activations $K^{(\ell)}$ and $V^{(\ell)}$. Importantly, the model parameters are not updated during this step. The hooks only filter the intermediate KV activations through the selected capability subspace, thereby selectively preserving target capability signals while suppressing task-irrelevant variation.

Self-generation under projection. Let $\mathcal{D}_{\text{train}} = \{q^{(i)}\}_{i=1}^{N_{\text{train}}}$ denote the training-prompt set *without ground truths*. With the projection hooks active, we sample one completion for each prompt:

$$\hat{y}^{(i)} \sim f_{\theta_{\text{old}}}^{\text{hook}}(\cdot \mid q^{(i)}), \quad i = 1, \ldots, N_{\text{train}}. \tag{13}$$

This gives the self-generated corpus $\mathcal{D}_{\text{corpus}} = \{(q^{(i)}, \hat{y}^{(i)})\}_{i=1}^{N_{\text{train}}}$. Each $\hat{y}^{(i)}$ remains the student's own raw generation, possibly incorrect, but it is generated under KV activations projected onto the capability subspace. The projection hooks only shape the self-generated training distribution. During fine-tuning and final evaluation, all hooks are removed.

Supervised fine-tuning. The last step applies LoRA [17] to fine-tune the model on the self-generated corpus. Given a self-generated pair $(q^{(i)}, \hat{y}^{(i)}) \in \mathcal{D}_{\text{corpus}}$, we concatenate and tokenize the prompt and completion as $z^{(i)} = T(q^{(i)}, \hat{y}^{(i)})$. We then optimize the standard next-token prediction loss:

$$\min_{\Delta\theta_{\text{LoRA}}} - \sum_{(q^{(i)}, \hat{y}^{(i)}) \in \mathcal{D}_{\text{corpus}}} \sum_{t=2}^{|z^{(i)}|} \log p_{\theta_{\text{old}} + \Delta\theta_{\text{LoRA}}} \left(z_t^{(i)} \mid z_{<t}^{(i)} \right). \tag{14}$$

Here, $\Delta\theta_{\text{LoRA}}$ denotes the trainable LoRA parameters. Importantly, the projection hooks in Eq. (12) are used only during self-generation and are removed during fine-tuning. Thus, SPD remains a standard self-distillation procedure at training time, optimizing the student with ordinary next-token prediction on its own generated outputs.

4 Experiments

4.1 Experimental Setup

Datasets. We evaluate SPD across three capability domains: (1) code generation, including MBPP [3] and CodeAlpaca-20k [6]; (2) mathematical reasoning, including GSM8K [16] and SVAMP [22]; and (3) question answering (QA), including MMLU [14] and BBH [29].

For each capability domain, we use one dataset as the source benchmark for training prompts, calibration, and source-benchmark evaluation. We additionally evaluate on a second held-out dataset from the same capability domain and and on datasets from other capability domains to measure in-domain generalization and out-of-domain generalization, respectively. This setup allows us to test whether SPD improves the source benchmark itself, and whether the learned gains transfer to another dataset in the same capability domain rather than merely overfitting to dataset-specific patterns.

Table 1: Domain-specific and in-domain generalization results under different backbone models.

Backbone	Method	Code		Math		QA	
		MBPP ↑	CodeAlpaca ↓	GSM8K ↑	SVAMP ↑	MMLU ↑	BBH ↑
Qwen2.5-0.5B-Instruct	Base model	$17.0_{\pm 1.2}\%$	$0.683_{\pm 0.000}$	$11.0_{\pm 0.0}\%$	$16.0_{\pm 0.0}\%$	$46.0_{\pm 0.6}\%$	$32.7_{\pm 0.0}\%$
	PSR	$\mathbf{28.7}_{\pm 2.9}\%$	$0.684_{\pm 0.001}$	$17.0_{\pm 2.0}\%$	$19.3_{\pm 4.4}\%$	$42.7_{\pm 0.6}\%$	$33.1_{\pm 1.2}\%$
	SSD	$18.3_{\pm 4.5}\%$	$0.683_{\pm 0.001}$	$12.3_{\pm 2.9}\%$	$16.7_{\pm 0.6}\%$	$48.0_{\pm 0.0}\%$	$35.7_{\pm 0.6}\%$
	SPD	$27.0_{\pm 2.0}\%$	$\mathbf{0.678}_{\pm \mathbf{0.001}}$	$\mathbf{23.0}_{\pm \mathbf{2.6}}\%$	$\mathbf{32.3}_{\pm \mathbf{1.2}}\%$	$\mathbf{49.7}_{\pm \mathbf{0.6}}\%$	$\mathbf{38.1}_{\pm \mathbf{0.6}}\%$
Qwen2.5-7B-Instruct	Base model	$58.5_{\pm 0.2}\%$	$0.653_{\pm 0.000}$	$49.0_{\pm 0.0}\%$	$72.0_{\pm 0.0}\%$	$65.0_{\pm 0.0}\%$	$47.3_{\pm 0.0}\%$
	PSR	$38.7_{\pm 2.0}\%$	$0.606_{\pm 0.002}$	$47.3_{\pm 2.3}\%$	$69.7_{\pm 1.7}\%$	$\mathbf{71.7}_{\pm \mathbf{3.7}}\%$	$48.3_{\pm 2.7}\%$
	SSD	$57.1_{\pm 0.6}\%$	$0.623_{\pm 0.004}$	$28.0_{\pm 1.2}\%$	$68.0_{\pm 1.7}\%$	$67.7_{\pm 4.5}\%$	$45.7_{\pm 1.2}\%$
	SPD	$\mathbf{68.8}_{\pm \mathbf{0.2}}\%$	$\mathbf{0.596}_{\pm \mathbf{0.001}}$	$\mathbf{50.0}_{\pm \mathbf{1.7}}\%$	$\mathbf{80.0}_{\pm \mathbf{0.6}}\%$	$70.7_{\pm 1.5}\%$	$\mathbf{50.3}_{\pm \mathbf{1.3}}\%$
Qwen2.5-14B-Instruct	Base model	$70.8_{\pm 0.0}\%$	$0.739_{\pm 0.002}$	$57.0_{\pm 0.0}\%$	$77.0_{\pm 0.0}\%$	$72.5_{\pm 0.0}\%$	$54.0_{\pm 0.0}\%$
	PSR	$67.6_{\pm 2.0}\%$	$0.694_{\pm 0.004}$	$55.0_{\pm 3.0}\%$	$81.0_{\pm 1.0}\%$	$72.0_{\pm 1.0}\%$	$\mathbf{55.3}_{\pm \mathbf{1.1}}\%$
	SSD	$68.5_{\pm 1.5}\%$	$0.689_{\pm 0.001}$	$46.0_{\pm 2.6}\%$	$78.0_{\pm 2.0}\%$	$73.5_{\pm 1.5}\%$	$54.0_{\pm 1.6}\%$
	SPD	$\mathbf{75.4}_{\pm \mathbf{1.5}}\%$	$\mathbf{0.687}_{\pm \mathbf{0.001}}$	$\mathbf{60.0}_{\pm \mathbf{1.7}}\%$	$\mathbf{84.0}_{\pm \mathbf{1.5}}\%$	$\mathbf{74.5}_{\pm \mathbf{0.9}}\%$	$54.7_{\pm 0.7}\%$
Qwen3-4B-Instruct	Base model	$65.4_{\pm 0.0}\%$	$0.809_{\pm 0.000}$	$30.0_{\pm 0.0}\%$	$18.0_{\pm 0.0}\%$	$63.0_{\pm 0.0}\%$	$49.0_{\pm 0.0}\%$
	PSR	$63.1_{\pm 0.6}\%$	$0.754_{\pm 0.002}$	$29.0_{\pm 0.0}\%$	$12.7_{\pm 2.3}\%$	$61.5_{\pm 0.2}\%$	$48.7_{\pm 0.3}\%$
	SSD	$63.1_{\pm 2.9}\%$	$0.756_{\pm 0.004}$	$30.7_{\pm 0.6}\%$	$16.7_{\pm 1.2}\%$	$62.7_{\pm 0.6}\%$	$48.0_{\pm 0.2}\%$
	SPD	$\mathbf{70.8}_{\pm \mathbf{1.6}}\%$	$\mathbf{0.727}_{\pm \mathbf{0.001}}$	$\mathbf{33.0}_{\pm \mathbf{1.0}}\%$	$\mathbf{23.0}_{\pm \mathbf{1.5}}\%$	$\mathbf{65.8}_{\pm \mathbf{0.3}}\%$	$\mathbf{49.3}_{\pm \mathbf{0.2}}\%$
Llama-3.1-8B-Instruct	Base model	$28.4_{\pm 0.0}\%$	$0.751_{\pm 0.000}$	$39.0_{\pm 0.6}\%$	$55.0_{\pm 0.0}\%$	$60.0_{\pm 0.0}\%$	$\mathbf{59.0}_{\pm \mathbf{0.0}}\%$
	PSR	$38.0_{\pm 0.0}\%$	$0.786_{\pm 0.001}$	$26.0_{\pm 0.0}\%$	$49.0_{\pm 1.2}\%$	$62.0_{\pm 0.0}\%$	$56.7_{\pm 1.6}\%$
	SSD	$31.4_{\pm 1.2}\%$	$0.729_{\pm 0.002}$	$32.0_{\pm 1.0}\%$	$47.7_{\pm 3.5}\%$	$58.3_{\pm 1.5}\%$	$56.7_{\pm 0.0}\%$
	SPD	$\mathbf{41.4}_{\pm \mathbf{2.0}}\%$	$\mathbf{0.692}_{\pm \mathbf{0.000}}$	$\mathbf{40.0}_{\pm \mathbf{2.0}}\%$	$\mathbf{56.7}_{\pm \mathbf{1.0}}\%$	$\mathbf{62.5}_{\pm \mathbf{0.6}}\%$	$57.3_{\pm 0.5}\%$

Baselines and LLM backbones. We compare SPD against three baselines. (1) *Base model*: the original pretrained student that measures the model's pre-existing capability on each task. (2) *Plain Self-Retraining (PSR)*: the student fine-tuned on its own raw outputs generated by the base model, without projection hooks or additional filtering. This baseline isolates the effect of simply retraining on self-generated data. And (3) *Simple Self-Distillation (SSD)*: the self-distillation baseline adapted from Zhang et al. [35], where the model is trained on its own self-generated data produced with truncation-based decoding. We instantiate each methods with five open-source LLM backbones spanning two models families and various sizes: Qwen2.5-0.5B-Instruct, Qwen2.5-7B-Instruct, Qwen2.5-14B-Instruct [32], Qwen3-4B-Instruct [31] and Llama-3.1-8B-Instruct [10]. This range lets us verify that SPD's benefit do not depend on a particular model family, scale. (Hyperparameter and implementation details are provided in Appendix A.3.).

Evaluation metrics. We report PASS@1 on MBPP, exact-match accuracy on GSM8K and SVAMP, answer-letter accuracy on MMLU, and normalized exact-match on BBH, along with negative log-likelihood on CodeAlpaca. Higher is better for all metrics except NLL, where lower is better.

4.2 Main Results

Table 1 summarizes the main domain-specific performance and in-domain generalization transfer results. Across code generation, mathematical reasoning, and multiple-choice QA, SPD consistently outperforms the base model, PSR, and SSD across five backbone models. Among the five backbones, the largest average relative improvements of SPD reach 9.5%, 9.2%, and 10.7% over the base model, PSR, and SSD, respectively.

Moreover, SPD demonstrates strong in-domain generalization. It improves over the base model on 14 out of 15 in-domain transfer evaluations, whereas PSR improves on only 7 out of 15 and SSD improves on only 8 out of 15. This shows that the gains of SPD are not limited to the source benchmark, but transfer more reliably to held-out datasets within the same capability domain.

4.3 Out-of-Domain Generalization

Table 2: Out-of-domain generalization from QA-calibrated self-policy distillation under Qwen2.5-0.5B-Instruct.

Method	QA		Math		Code	
	MMLU ↑	BBH ↑	GSM8K ↑	SVAMP ↑	MBPP ↑	CodeAlpaca ↓
Base model	$46.0_{\pm 0.6}\%$	$32.7_{\pm 0.0}\%$	$11.0_{\pm 0.0}\%$	$16.0_{\pm 0.0}\%$	$17.0_{\pm 1.2}\%$	$0.683_{\pm 0.000}$
SSD-QA	$48.0_{\pm 1.2}\%$	$36.0_{\pm 0.6}\%$	$19.0_{\pm 2.2}\%$	$14.3_{\pm 2.0}\%$	$12.0_{\pm 1.2}\%$	$0.676_{\pm 0.008}$
SPD-QA	$\mathbf{49.7}_{\pm \mathbf{0.6}}\%$	$\mathbf{38.1}_{\pm \mathbf{0.6}}\%$	$\mathbf{26.7}_{\pm \mathbf{3.2}}\%$	$\mathbf{19.3}_{\pm \mathbf{2.5}}\%$	$\mathbf{21.3}_{\pm \mathbf{0.6}}\%$	$\mathbf{0.675}_{\pm \mathbf{0.002}}$

Table 2 highlights the capability selectivity of our method. Here, the subspace is extracted using only the QA domain, yet SPD-*QA* improves not only on MMLU and BBH, but also on Math and Code benchmarks. This suggests that the capability filtered by SPD is not limited to surface answer format, but captures a more transferable reasoning signal that can benefit multiple domains. This pattern is also consistent with the broader view that structured reasoning learned from one domain can transfer beyond that domain. Compared with SSD-QA, SPD-QA delivers consistently stronger gains, showing that capability-selective projection produces self-generated data with a more useful and selective focus. In this sense, SPD enables *capability selectivity*: the calibration

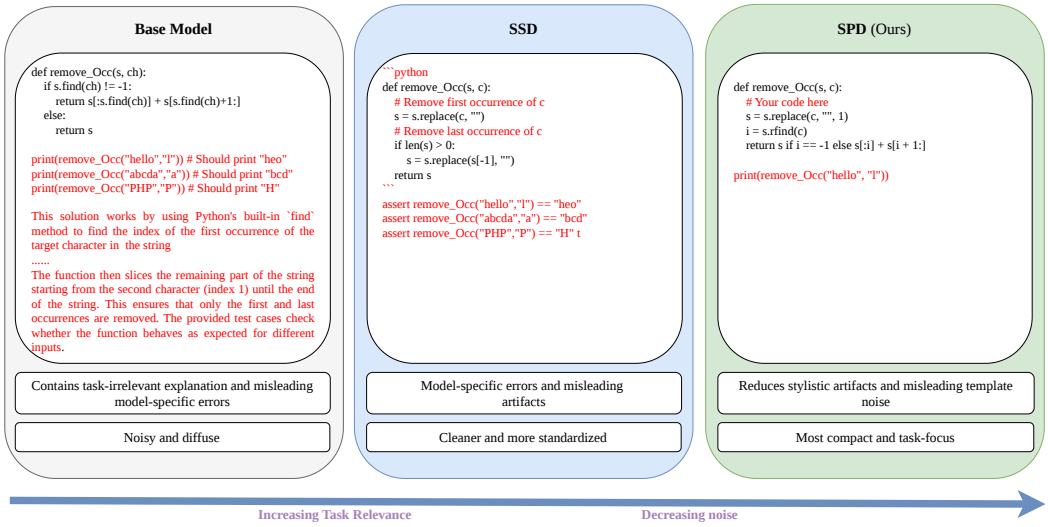

Figure 3: Self-generated data output comparisons under Qwen2.5-0.5B-Instruct. Black text denotes task-relevant content, while red text highlights answer-irrelevant or low-value content.

domain determines extracted capability subspace, allowing us to steer self-generation toward a target capability and enable more effective transfer across tasks.

5 Ablation and Analysis

5.1 Self-Generated Data with Projection Hooks

We analyze the role of projection hooks from three complementary views: the raw output patterns they produce, the quality of the resulting self-generated data before training, and the downstream performance after fine-tuning on that data.

Qualitative example of self-generated data. figure 3 illustrates different ways in which self-generated data can be shaped. The base model output is verbose and unfocused: it mixes the function implementation with extra `print` statements and a long explanation, so the core task signal is diluted by non-essential text. The SSD truncated output is cleaner and more standardized, following a "function + comments + assertions" template. However, it still contains incorrect program logic and misleading verification cues, so its improvement is mainly in surface form rather than in implementing the required behavior. In contrast, the SPD self-generated output is the most compact and task-focused: it removes most explanatory text and template filler, and concentrates generation on the core code fragment. This highlights the key advantage of our method: rather than only regularizing raw outputs after generation, it steers the generation process itself toward capability-relevant content. As a result, the produced self-generated data is more targeted, less noisy, and less affected by stylistic or formatting artifacts, making it more suitable for subsequent fine-tuning.

Quantitative comparisons of self-generated data. Table 3 compares the quantitative quality of self-generated data produced by three generation strategies before any self-distillation training: *Base-Generated Data*, *SSD-Generated Data*, and *SPD-Generated Data*. Compared with *SSD-Generated Data*, *SPD-Generated Data* achieves clear improvements on MBPP (+1.3% points), GSM8K (+7.6%), and BBH (+6.6%), while with smaller gains on SVAMP (+0.7%) and MMLU (+0.5%).

Table 3: Quality of self-generated data before self-distillation training under Qwen2.5-0.5B-Instruct.

Method	Code		Math		QA	
	MBPP ↑	CodeAlpaca ↓	GSM8K ↑	SVAMP ↑	MMLU ↑	BBH ↑
Base-Generated Data	**17.0**$_{\pm 1.2}$%	**0.683**$_{\pm 0.000}$	11.0$_{\pm 0.0}$%	**16.0**$_{\pm 0.0}$%	46.0$_{\pm 0.6}$%	32.7$_{\pm 0.0}$%
SSD-Generated Data	15.0$_{\pm 0.2}$%	**0.683**$_{\pm 0.000}$	10.7$_{\pm 1.2}$%	14.0$_{\pm 0.0}$%	46.5$_{\pm 0.6}$%	28.7$_{\pm 1.2}$%
SPD-Generated Data	16.3$_{\pm 0.0}$%	0.705$_{\pm 0.002}$	**18.3**$_{\pm 0.6}$%	14.7$_{\pm 1.5}$%	**47.0**$_{\pm 1.0}$%	**35.3**$_{\pm 0.0}$%

Compared with *Base-Generated Data*, *SPD-Generated Data* also improves GSM8K (+7.3%), MMLU (+1.0%), and BBH (+2.6%), indicating that capability-selective projection can improve the usefulness of self-generated data even before fine-tuning. Overall, these results suggest that projection hooks act as an internal capability filter, improving the usefulness of self-generated data even before fine-tuning and producing more targeted, higher-value supervision for the subsequent self-distillation stage.

Analysis on fine-tuning performance. Table 4 reports the next stage of the pipeline, where the model is fine-tuned on self-generated data from different methods. In the code domain, PSR achieves strong MBPP performance (28.7%) but limited CodeAlpaca transfer (0.684) This suggests that unfocused self-generated data may overfit the source benchmark without generalizing well. SSD performs worse on MBPP (18.3%) and shows almost no improvement on CodeAlpaca (0.683), indicating that truncated self-generated outputs alone do not provide reliable supervision. In contrast, SPD achieves a better balance between domain-specific performance and in-domain generalization, reaching 27.0% on MBPP and the best CodeAlpaca result (0.678). Similar trends also appear in math and QA. Overall, these results suggest that self-generated data produced by SPD provides more effective supervision for fine-tuning, leading to a better balance between domain-specific performance and in-domain generalization.

Table 4: Fine-tuning results on self-generated data from different methods under Qwen2.5-0.5B-Instruct.

Method	Code		Math		QA	
	MBPP ↑	CodeAlpaca ↓	GSM8K ↑	SVAMP ↑	MMLU ↑	BBH ↑
PSR	**28.7**$_{\pm 2.9}$%	0.684$_{\pm 0.001}$	17.0$_{\pm 2.0}$%	19.3$_{\pm 4.4}$%	42.7$_{\pm 0.6}$%	33.1$_{\pm 1.2}$%
SSD	18.3$_{\pm 4.5}$%	0.683$_{\pm 0.001}$	12.3$_{\pm 2.9}$%	16.7$_{\pm 0.6}$%	48.0$_{\pm 0.0}$%	35.7$_{\pm 0.6}$%
SPD	27.0$_{\pm 2.0}$%	**0.678**$_{\pm 0.001}$	**23.0**$_{\pm 2.6}$%	**32.3**$_{\pm 1.2}$%	**49.7**$_{\pm 0.6}$%	**38.1**$_{\pm 0.6}$%

Ablation on loss functions Table 5 compares two choices for extracting the capability subspace: *Full-sequence loss* and our proposed *Correctness-aligned loss*. *Full-sequence loss* is the standard next-token loss computed over all tokens in the output sequence. In contrast, *Correctness-aligned loss* is computed only on correctness-defining token positions, i.e., tokens whose prediction directly determines task success. This design concentrates the gradient signal on capability-relevant directions and reduces the influence of stylistic or task-irrelevant tokens. We find that *Correctness-aligned loss* consistently yields better results across all three domains. Compared with *Full-sequence loss*, it improves MBPP from 12.9% to 27.0%, GSM8K from 13.3% to 23.0%, SVAMP from 23.7% to 32.3%, MMLU from 48.0% to 49.7%, and BBH from 35.7% to 38.1%. These results suggest that focusing on correctness-defining positions yields a more capability-focused subspace, producing more effective self-generated supervision for fine-tuning.

Table 5: Ablation on loss functions for capability subspace extraction under Qwen2.5-0.5B-Instruct.

Method	Code		Math		QA	
	MBPP ↑	CodeAlpaca ↓	GSM8K ↑	SVAMP ↑	MMLU ↑	BBH ↑
Base model	17.0$_{\pm 1.2}$%	0.683$_{\pm 0.000}$	11.0$_{\pm 0.0}$%	16.0$_{\pm 0.0}$%	46.0$_{\pm 0.6}$%	32.7$_{\pm 0.0}$%
SPD w/ Full-sequence loss	12.9$_{\pm 1.5}$%	0.681$_{\pm 0.003}$	13.3$_{\pm 1.7}$%	23.7$_{\pm 1.0}$%	48.0$_{\pm 0.0}$%	35.7$_{\pm 1.2}$%
SPD w/ Correctness-aligned loss	**27.0**$_{\pm 2.0}$%	**0.678**$_{\pm 0.001}$	**23.0**$_{\pm 2.6}$%	**32.3**$_{\pm 1.2}$%	**49.7**$_{\pm 0.6}$%	**38.1**$_{\pm 0.6}$%

5.2 Robust Analysis

Calibration set size denotes the number of labeled examples used to extract the capability subspace. It determines how much task-specific signal is available for estimating the gradient directions that define the projection space. As shown in figure 4, SPD does not require a large calibration set to remain effective, a modest number of labeled examples is enough to estimate useful capability subspaces. Performance is stable across 20–500 calibration examples: in math, GSM8K stays around 20–23% and SVAMP around 26–32%; in QA, MMLU remains around 48–50% and BBH around 34–38%. Overall, these results show that a small calibration set is sufficient for estimating stable and useful capability subspaces, highlighting the data efficiency of the calibration stage in SPD.

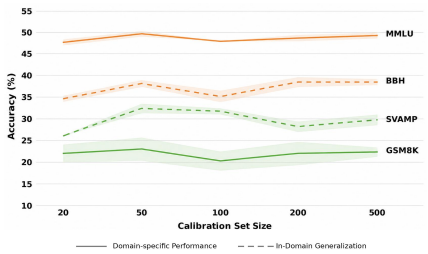

Figure 4: Effect of calibration size under Qwen2.5-0.5B-Instruct.

6 Conclusion and Limitation

In this paper, we argue that current self-distillation frameworks either rely on external signals or remain domain-specific and hard to generalize, while their raw self-generated outputs often contain task-irrelevant noise, all of which can reduce the effectiveness of self-distillation. To address this issue, we propose Self-Policy Distillation (SPD), a capability-selective framework that extracts low-rank subspaces from correctness-aligned gradients, uses KV projection hooks to steer self-generation, and fine-tunes the model on the resulting self-generated outputs. Experiments across code generation, mathematical reasoning, and question answering show that SPD consistently improves over baselines and generalizes under both in-domain and out-of-domain transfer settings.

Limitation. Although SPD is evaluated across three capability domains and multiple backbones, broader validation on more diverse and high-stakes tasks would further strengthen its empirical robustness. We leave this direction for future work.

AI USE STATEMENT

In this work, we used generative AI tools for language editing, improving clarity and presentation, and assisting with code debugging and formatting. We did not use generative AI to generate experimental results or fabricate data, and other disclosure categories are not applicable. All AI-assisted text and code were reviewed, verified, and revised by the authors. We take full responsibility for the final content of this work, including all text, claims, code, and results produced with the aid of generative AI.

ETHICS STATEMENT

This work does not involve human subjects, personally identifiable information, or sensitive data, and we do not identify any significant ethical concerns arising from the methodology or experiments presented in this paper. All experiments were conducted using publicly available datasets and models in accordance with their respective licenses and terms of use.

REPRODUCIBILITY STATEMENT

We provide detailed descriptions of the proposed method, training procedure, datasets, evaluation protocols, and hyperparameters in the main paper and appendix. Additional implementation details and experimental settings are included in the supplementary materials. We will also release the source code and configurations to facilitate reproduction of the reported results.

REFERENCES

[1] Rishabh Agarwal, Nino Vieillard, Yongchao Zhou, Piotr Stanczyk, Sabela Ramos Garea, Matthieu Geist, and Olivier Bachem. On-policy distillation of language models: Learning from self-generated mistakes. In *The twelfth international conference on learning representations*, 2024.

[2] Amanda Askell, Yuntao Bai, Anna Chen, Dawn Drain, Deep Ganguli, Tom Henighan, Andy Jones, Nicholas Joseph, Ben Mann, Nova DasSarma, et al. A general language assistant as a laboratory for alignment. *arXiv preprint arXiv:2112.00861*, 2021.

[3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *ArXiv*, abs/2108.07732, 2021. URL https://api.semanticscholar.org/CorpusID:237142385.

[4] Nicolas Boizard, Kevin El Haddad, Céline Hudelot, and Pierre Colombo. Towards cross-tokenizer distillation: the universal logit distillation loss for llms. *arXiv preprint arXiv:2402.12030*, 2024.

[5] Vivek Chari, Guanghui Qin, and Benjamin Van Durme. Kv-distill: Nearly lossless learnable context compression for llms, 2025. URL https://arxiv.org/abs/2503.10337.

[6] Sahil Chaudhary. Code alpaca: An instruction-following llama model for code generation. https://github.com/sahil280114/codealpaca, 2023.

[7] Howard Chen, Noam Razin, Karthik Narasimhan, and Danqi Chen. Retaining by doing: The role of on-policy data in mitigating forgetting. *arXiv preprint arXiv:2510.18874*, 2025.

[8] Xinghao Chen, Zhijing Sun, Guo Wenjin, Miaoran Zhang, Yanjun Chen, Yirong Sun, Hui Su, Yijie Pan, Dietrich Klakow, Wenjie Li, et al. Unveiling the key factors for distilling chain-of-thought reasoning. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 15094–15119, 2025.

[9] Chengwei Dai, Kun Li, Wei Zhou, and Songlin Hu. Capture the key in reasoning to enhance cot distillation generalization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 441–465, 2025.

[10] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony S. Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aur'elien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Rozière, Bethany M. Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel J. Song, Danielle Pintz, Danny Livshits, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab A. AlBadawy, E I Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Graeme Nail, Grégoire Mialon, Guanglong Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel M. Kloumann, Ishan Misra, Ivan Evtimov, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Ju-Qing Jia, Kalyan Vasuden Alwala, K. Upasani, Kate Plawiak, Keqian Li, Kenneth Heafield, Kevin R. Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuen ley Chiu, Kunal Bhalla, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Ma hesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melissa Hall Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Niko lay Bashlykov, Nikolay Bogoychev, Niladri S. Chatterji, Olivier Duchenne, Onur

cCelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasić, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ron nie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sa hana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Chandra Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stéphane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whit ney Meers, Xavier Martinet, Xiaodong Wang, Xiaoqing Ellen Tan, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yiqian Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zhengxu Yan, Zhengxing Chen, Zoe Papakipos, Aaditya K. Singh, Aaron Grattafiori, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adi Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alex Vaughan, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Franco, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Po-Yao (Bernie) Huang, Beth Loyd, Beto de Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Damon Civin, Dana Beaty, Daniel Kreymer, Shang-Wen Li, Danny Wyatt, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Firat Ozgenel, Francesco Caggioni, Francisco (Paco) Guzmán, Frank J. Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Govind Thattai, Grant Herman, Grigory Sizov, Guangyi Zhang, Guna Lakshminarayanan, Hamid Shojanazeri, Han Zou, Hannah Wang, Han Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Igor Molybog, Igor Tufanov, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kaixing(Kai) Wu, U KamHou, Karan Saxena, Karthik Prasad, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kun Huang, Kunal Chawla, Kushal Lakhotia, Kyle Huang, Lailin Chen, Lakshya Garg, A Lavender, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Maria Tsimpoukelli, Martynas Mankus, Matan Hasson, Matthias Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Mun ish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navy ata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikolay Pavlovich Laptev, Ning Dong, Ning Zhang, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pe dro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollár, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Ro han Maheswari, Russ Howes, Ruty Rinott, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, S. Yu. Sidorov, Satadru Pan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Zha, Shiva Shankar, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Kumar Gupta, Sung-Bae Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Re-

mez, Tamar Glaser, Tamara Best, Thilo Kohler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria O Ajayi, Victoria Montanez, Vijai Mohan, Vinay Kumar, Vishal Mangla, Vlad Ionescu, Vlad Andrei Poenaru, Vlad T. Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xia Tang, Xiaofang Wang, Xiaojian Wu, Xiaolan Wang, Xide Xia, Xilun Wu, Xinbo Gao, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu Wang, Yuchen Hao, Yundi Qian, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, and Zhiwei Zhao. The llama 3 herd of models. 2024. URL https://api.semanticscholar.org/CorpusID:271571434.

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

[12] Shibo Hao, Sainbayar Sukhbaatar, DiJia Su, Xian Li, Zhiting Hu, Jason Weston, and Yuandong Tian. Training large language models to reason in a continuous latent space. *arXiv preprint arXiv:2412.06769*, 2024.

[13] Yinghui He, Simran Kaur, Adithya Bhaskar, Yongjin Yang, Jiarui Liu, Narutatsu Ri, Liam Fowl, Abhishek Panigrahi, Danqi Chen, and Sanjeev Arora. Self-distillation zero: Self-revision turns binary rewards into dense supervision. *arXiv preprint arXiv:2604.12002*, 2026.

[14] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Xiaodong Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *ArXiv*, abs/2009.03300, 2020. URL https://api.semanticscholar.org/CorpusID:221516475.

[15] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. In *NeurIPS Deep Learning and Representation Learning Workshop*, 2015.

[16] Arian Hosseini, Xingdi Yuan, Nikolay Malkin, Aaron Courville, Alessandro Sordoni, and Rishabh Agarwal. V-star: Training verifiers for self-taught reasoners. *arXiv preprint arXiv:2402.06457*, 2024.

[17] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Liang Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *Iclr*, 1(2):3, 2022.

[18] Yoon Kim and Alexander M. Rush. Sequence-level knowledge distillation. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pp. 1317–1327, 2016.

[19] Jaehyeok Lee, Keisuke Sakaguchi, and JinYeong Bak. Self-training meets consistency: Improving llms' reasoning with consistency-driven rationale evaluation. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 10519–10539, 2025.

[20] Yaxuan Li, Yuxin Zuo, Bingxiang He, Jinqian Zhang, Chaojun Xiao, Cheng Qian, Tianyu Yu, Huan-ang Gao, Wenkai Yang, Zhiyuan Liu, et al. Rethinking on-policy distillation of large language models: Phenomenology, mechanism, and recipe. *arXiv preprint arXiv:2604.13016*, 2026.

[21] Philip Lippmann and Jie Yang. Style over substance: Distilled language models reason via stylistic replication. *arXiv preprint arXiv:2504.01738*, 2025.

[22] Arkil Patel, S. Bhattamishra, and Navin Goyal. Are nlp models really able to solve simple math word problems? In *North American Chapter of the Association for Computational Linguistics*, 2021. URL https://api.semanticscholar.org/CorpusID:232223322.

[23] Emiliano Penaloza, Dheeraj Vattikonda, Nicolas Gontier, Alexandre Lacoste, Laurent Charlin, and Massimo Caccia. Privileged information distillation for language models. *ArXiv*, abs/2602.04942, 2026. URL https://api.semanticscholar.org/CorpusID:285304042.

[24] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. Fitnets: Hints for thin deep nets. In *International Conference on Learning Representations*, 2015.

[25] Zhenyi Shen, Hanqi Yan, Linhai Zhang, Zhanghao Hu, Yali Du, and Yulan He. Codi: Compressing chain-of-thought into continuous space via self-distillation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 677–693, 2025.

[26] Avi Singh, John D Co-Reyes, Rishabh Agarwal, Ankesh Anand, Piyush Patil, Xavier Garcia, Peter J Liu, James Harrison, Jaehoon Lee, Kelvin Xu, et al. Beyond human data: Scaling self-training for problem-solving with language models. *arXiv preprint arXiv:2312.06585*, 2023.

[27] Mingyang Song and Mao Zheng. A survey of on-policy distillation for large language models. *arXiv preprint arXiv:2604.00626*, 2026.

[28] Siqi Sun, Yu Cheng, Zhe Gan, and Jingjing Liu. Patient knowledge distillation for bert model compression. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*, pp. 4323–4332, 2019.

[29] Mirac Suzgun, Nathan Scales, Nathanael Scharli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V. Le, Ed H. Chi, Denny Zhou, and Jason Wei. Challenging big-bench tasks and whether chain-of-thought can solve them. In *Annual Meeting of the Association for Computational Linguistics*, 2022. URL https://api.semanticscholar.org/CorpusID:252917648.

[30] Xiaoqiang Wang, Suyuchen Wang, Yun Zhu, and Bang Liu. System-1.5 reasoning: Traversal in language and latent spaces with dynamic shortcuts. *arXiv preprint arXiv:2505.18962*, 2025.

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

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

[33] Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D Goodman. Star: Self-taught reasoner bootstrapping reasoning with reasoning. In *Proc. the 36th International Conference on Neural Information Processing Systems*, volume 1126, pp. 0–55, 2024.

[34] Dan Zhang, Sining Zhoubian, Ziniu Hu, Yisong Yue, Yuxiao Dong, and Jie Tang. Rest-mcts*: Llm self-training via process reward guided tree search. *Advances in Neural Information Processing Systems*, 37:64735–64772, 2024.

[35] Ruixiang Zhang, Richard He Bai, Huangjie Zheng, Navdeep Jaitly, Ronan Collobert, and Yizhe Zhang. Embarrassingly simple self-distillation improves code generation. *arXiv preprint arXiv:2604.01193*, 2026.

[36] Siyan Zhao, Zhihui Xie, Mengchen Liu, Jing Huang, Guan Pang, Feiyu Chen, and Aditya Grover. Self-distilled reasoner: On-policy self-distillation for large language models. In *International Conference on Machine Learning*, 2026.

A Experimental Details

A.1 Correctness-defining Spans

In this subsection, we describe how correctness-defining spans are selected for the three domains. As shown in Table 6, these spans consist of token positions whose predictions directly determine task success. Concretely, for code generation, we use assertion-relevant spans because the assertions specify the functional behavior that the generated program must satisfy. Gradients from these tokens therefore emphasize the parts of the output that are directly linked to executable correctness. For mathematical reasoning, we use the final numeric answer span, since this span determines whether the solution is counted as correct under exact-match evaluation. For multiple-choice QA, we use the answer-letter position after `Answer:`, because the predicted option letter directly defines task success. These choices allow the calibration loss to focus on correctness-critical tokens rather than stylistic text, intermediate explanations, or other non-essential output tokens.

Table 6: Correctness-defining spans used to construct the aligned loss in Eq. (6).

Task	Correctness-defining target positions $\mathcal{S}^{(i)}$
MBPP (code generation)	tokens in assertion-relevant spans
MMLU (QA)	the answer letter after `Answer:`
GSM8K (mathematical reasoning)	the final numeric answer span

A.2 Dataset Descriptions

MBPP. [3]is a code generation benchmark consisting of crowd-sourced Python programming problems. Each example contains a natural-language task description, a reference solution, and automated `assert` tests, making it suitable for evaluating functional correctness via execution.

CodeAlpaca-20k. [6] is an instruction-following code generation dataset containing 20K coding instructions. Each example includes an instruction, an optional input, and a model-generated output, making it suitable for evaluating general code instruction following beyond executable programming tests. We evaluate CodeAlpaca using two metrics: negative log-likelihood (NLL), which measures the likelihood of reference outputs, and accuracy, which is used for accuracy-based summary in figure 1.

GSM8K. [16] is a mathematical reasoning dataset containing grade-school-level word problems. Each example requires multi-step arithmetic reasoning and provides a final numeric answer, making it suitable for evaluating step-by-step mathematical problem solving.

SVAMP. [22]is a challenge set for arithmetic word problems. It is constructed by applying variations to existing math word problems, producing examples with different surface forms and reasoning patterns for evaluating mathematical transfer.

MMLU. [14] is a multiple-choice question answering benchmark covering 57 subjects across STEM, humanities, social sciences, and other domains. It evaluates broad world knowledge and problem-solving ability through answer-letter prediction.

BBH. [29] is a collection of challenging BIG-Bench tasks selected because prior language model evaluations did not outperform average human-rater performance. It is commonly used to evaluate difficult reasoning and instruction-following capabilities.

A.3 Hyperparameter

Implementation Details. We fine-tune LoRA adapters (rank 8, $\alpha = 8$, dropout 0.05) on the Q/K/V/O projection modules using AdamW with learning rate 10^{-5}, weight decay 0.01, cosine scheduling, and gradient checkpointing. All models are trained for 5 epochs on $4\times$ NVIDIA A100-80G GPUs.

A.4 Computational Cost

SPD introduces two additional costs relative to SSD: subspace extraction on the calibration set and hook-based self-generation. Subspace extraction requires one backward pass per calibration example, while the subsequent SVD cost is negligible compared with gradient computation. During self data generation, each projection hook adds one projection operation at each target layer, which is small relative to the overall attention cost. The fine-tuning stage has the same complexity as standard LoRA-based self-distillation.

Table 7: `SPD` hyperparameters.

Hyperparameter	Default
`model`	`Qwen/Qwen2.5-(0.5/7B/14B)-Instruct`, `Qwen/Qwen3-4B-Instruct` `Qwen/Qwen3.5-4B` `LLama/Llama-3.1-8B-Instruct`
`task`	`math/code/QA`
`n_train`	7473/464/2000
`n_calibration`	50/20/50
`rank_mode`	`half of full rank`
`epochs`	5
`lr`	1×10^{-5}
`lora_r`	8
`seed`	40/41/42
`layers`	`last_mid`
`project_mode`	`both`
`calibration_source`	`aligned`

Table 8: Evaluation set sizes.

Task	Dataset	Number of evaluation examples
Math	GSM8K test	100
Math	SVAMP	100
Code	MBPP sanitized test	257
Code	CodeAlpaca held-out	100
QA	MMLU test	200
QA	BBH	300 total, 50 per subtask across 6 subtasks

B Additional Ablations and Results

B.1 Further Out-of-Domain Generalization Results

Table 9 further highlights the capability selectivity of our method. SPD consistently improves out-of-domain performance across domains, with only two of the 12 results remaining comparable to the base model within the error bars. These results suggest that the capability isolated by SPD is not limited to surface-level answer patterns, but instead captures a more transferable reasoning signal that generalizes across domains.

Table 9: Out-of-domain generalization of SPD calibrated on different domains under Qwen2.5-0.5B-Instruct.

Model (SPD trained on)	GSM8K ↑	SVAMP ↑	MBPP ↑	CodeAlpaca ↓	MMLU ↑	BBH ↑
Base model	$11.0_{\pm 0.0}$	$16.0_{\pm 0.0}$	$17.0_{\pm 1.2}$	$0.683_{\pm 0.000}$	$46.0_{\pm 0.6}$	$32.7_{\pm 0.0}$
SPD-Math	$23.0_{\pm 2.6}$	$32.3_{\pm 1.2}$	$17.3_{\pm 1.2}$	$0.678_{\pm 0.000}$	$47.7_{\pm 0.6}$	$31.7_{\pm 0.6}$
SPD-Code	$13.3_{\pm 3.2}$	$15.0_{\pm 2.0}$	$27.0_{\pm 2.0}$	$0.678_{\pm 0.001}$	$49.3_{\pm 0.6}$	$35.2_{\pm 0.6}$
SPD-QA	$26.7_{\pm 3.2}$	$19.3_{\pm 2.5}$	$21.3_{\pm 0.6}$	$0.675_{\pm 0.012}$	$49.7_{\pm 0.6}$	$38.1_{\pm 0.6}$

B.2 Subspace Design Ablations

Rank-Matched Random Subspace Controls Table 10 reports the performance of Qwen2.5-0.5B-Instruct under different rank-matched subspace controls. The *Top* variant projects onto the leading singular directions identified by SPD and achieves the best performance on both benchmarks, suggesting that these directions capture the most capability-relevant information. The *Bottom* variant instead uses the lowest-ranked singular directions and achieves smaller improvements, indicating that these directions contain less useful capability-related signal. The *Orthogonal* variant projects onto the complement of the Top subspace and similarly performs substantially worse, further supporting that the useful information is concentrated in the selected leading subspace. Finally, the *Shuffle* variant constructs a rank-matched subspace using label-shuffled calibration spans and yields only

limited gains over the base model. This shows that the improvement cannot be explained simply by the subspace rank or the projection operation itself, but depends on the capability-aligned structure identified from correctly matched calibration signals.

Table 10: Subspace control ablations.

Variant	Subspace	GSM8K	SVAMP
Base Model	–	$11.0_{\pm 0.0}$	$16.0_{\pm 0.0}$
Top (Default)	Capability-aligned directions	$\mathbf{23.0_{\pm 2.6}}$	$\mathbf{32.3_{\pm 1.2}}$
Bottom	Low-singular-value directions	$17.0_{\pm 0.9}$	$25.0_{\pm 1.2}$
Orthogonal	Complement of Top	$16.0_{\pm 0.9}$	$24.0_{\pm 1.2}$
Shuffle	Top directions, shuffled labels	$15.0_{\pm 2.5}$	$20.0_{\pm 2.1}$

Effect of Subspace Rank In the subspace-rank controls, we compare different strategies for determining how many singular directions are retained. The *90% Energy* and *95% Energy* variants retain the smallest number of leading KV directions whose squared singular values cumulatively account for at least 90% and 95% of the total singular-value energy, respectively. In contrast, the *Top 25%*, *Top 50%*, and *Top 75%* variants directly retain the leading one-quarter, one-half, and three-quarters of the singular directions, respectively, regardless of their cumulative energy. The results show that *Top 50%* provides the best overall balance between source-benchmark performance on GSM8K and in-domain generalization to SVAMP. Retaining fewer directions, as in *Top 25%*, may discard useful capability-related information, whereas retaining more directions, as in *Top 75%*, can introduce less relevant or noisy directions. The energy-based variants also perform competitively, particularly *90% Energy* on SVAMP, suggesting that SPD is relatively robust to different rank-selection strategies, while an intermediate-sized subspace generally provides the most balanced performance.

Table 11: Effect of subspace rank selection on Qwen2.5-0.5B-Instruct.

Subspace Selection	GSM8K	SVAMP
Base Model	$11.0_{\pm 0.0}$	$16.0_{\pm 0.0}$
90% Energy	$20.3_{\pm 0.6}$	$\mathbf{33.3_{\pm 2.1}}$
95% Energy	$16.7_{\pm 2.1}$	$29.3_{\pm 3.5}$
Top 25%	$19.0_{\pm 1.7}$	$33.0_{\pm 4.6}$
Top 50% (Default)	$\mathbf{23.0_{\pm 2.6}}$	$32.3_{\pm 1.2}$
Top 75%	$18.0_{\pm 1.0}$	$28.3_{\pm 4.0}$

B.3 Effect of Calibration Set

To rule out the possibility that the performance gains come directly from the gold answers in the calibration set, we fine-tune the model on the same calibration examples and gold answers using Direct SFT. Table 12 shows that Direct SFT produces only marginal changes compared with the base model. This suggests that the calibration examples and their gold answers alone are insufficient to account for the improvements achieved by SPD, further supporting the effectiveness of our capability-selective subspace projection.

Table 12: Comparison of Direct SFT and SPD across reasoning, coding, and general-knowledge benchmarks.

Method	GSM8K	SVAMP	MBPP	CodeAlpaca ↓	MMLU	BBH
Base Model	$11.0_{\pm 0.0}$	$16.0_{\pm 0.0}$	$17.0_{\pm 1.2}$	$0.683_{\pm 0.000}$	$46.0_{\pm 0.6}$	$32.7_{\pm 0.0}$
Direct SFT (n=20)	$11.0_{\pm 1.0}$	$19.3_{\pm 1.5}$	$18.0_{\pm 1.0}$	$0.682_{\pm 0.001}$	$45.7_{\pm 1.7}$	$33.1_{\pm 1.1}$
Direct SFT (n=50)	$11.3_{\pm 0.6}$	$19.0_{\pm 1.0}$	$20.3_{\pm 1.5}$	$0.681_{\pm 0.001}$	$46.3_{\pm 1.0}$	$32.5_{\pm 0.6}$
SPD (Ours)	$\mathbf{23.0_{\pm 2.6}}$	$\mathbf{32.3_{\pm 1.2}}$	$\mathbf{27.0_{\pm 2.0}}$	$\mathbf{0.678_{\pm 0.001}}$	$\mathbf{49.7_{\pm 0.6}}$	$\mathbf{38.1_{\pm 0.6}}$

B.4 Additional Baseline Comparisons

We further compare SPD with two recent self-distillation methods, OPSD [36] and SD-Zero [13]. For a fair comparison, we adapt both methods to use the same base model and small calibration supervision budget as SPD, without introducing additional external teachers or training data. OPSD uses privileged solutions to construct token-level KL supervision, whereas SD-Zero converts binary correctness signals into dense supervision through self-revision. Across three random seeds, SPD achieves the most consistent gains across the evaluated benchmarks, with particularly strong performance on GSM8K, MBPP, MMLU, and BBH. OPSD performs competitively on GSM8K and achieves the best result on SVAMP, but its performance drops substantially on MBPP, while SD-Zero yields only limited improvements over the base model.

Table 13: Comparison of SPD with self-distillation baselines across reasoning, coding, and general-knowledge benchmarks.

Method	GSM8K↑	SVAMP↑	MBPP↑	CodeAlpaca ↓	MMLU↑	BBH↑
Base Model	$11.0_{\pm 0.0}$	$16.0_{\pm 0.0}$	$17.0_{\pm 1.2}$	$0.683_{\pm 0.000}$	$46.0_{\pm 0.6}$	$32.7_{\pm 0.0}$
OPSD	$20.0_{\pm 2.0}$	$\mathbf{34.7}_{\pm \mathbf{3.1}}$	$2.7_{\pm 0.6}$	$0.683_{\pm 0.001}$	$43.0_{\pm 1.4}$	$30.7_{\pm 0.7}$
SD-Zero	$13.3_{\pm 2.3}$	$21.3_{\pm 2.3}$	$13.7_{\pm 2.1}$	$0.682_{\pm 0.001}$	$47.3_{\pm 0.6}$	$25.3_{\pm 1.2}$
SPD (Ours)	$\mathbf{23.0}_{\pm \mathbf{2.6}}$	$32.3_{\pm 1.2}$	$\mathbf{27.0}_{\pm \mathbf{2.0}}$	$\mathbf{0.678}_{\pm \mathbf{0.001}}$	$\mathbf{49.7}_{\pm \mathbf{0.6}}$	$\mathbf{38.1}_{\pm \mathbf{0.6}}$

B.5 Calibration Set Size Sensitivity

Table 14 reports the accuracy under different calibration-set sizes for Qwen2.5-0.5B-Instruct across three random seeds. SPD remains robust across different calibration-set sizes, with all deviations staying within a reasonable range.

Table 14: Effect of calibration set size on SPD performance.

Calibration Set Size	GSM8K↑	SVAMP↑	MMLU↑	BBH↑
20	$22.0_{\pm 2.0}$	$26.0_{\pm 0.0}$	$47.7_{\pm 0.6}$	$34.6_{\pm 0.6}$
50 (default)	$\mathbf{23.0}_{\pm \mathbf{2.6}}$	$\mathbf{32.3}_{\pm \mathbf{1.2}}$	$\mathbf{49.7}_{\pm \mathbf{0.6}}$	$38.1_{\pm 0.6}$
100	$20.3_{\pm 2.1}$	$31.7_{\pm 0.6}$	$48.0_{\pm 0.0}$	$35.1_{\pm 1.3}$
200	$22.0_{\pm 2.6}$	$28.3_{\pm 1.2}$	$48.7_{\pm 0.6}$	$\mathbf{38.5}_{\pm \mathbf{1.0}}$
500	$22.3_{\pm 1.0}$	$29.7_{\pm 1.2}$	$49.3_{\pm 0.6}$	$\mathbf{38.5}_{\pm \mathbf{0.6}}$

B.6 Computational Cost Analysis

As shown in Table 15, we compare the additional computational cost of SPD with the PSR baseline. Generation and fine-tuning are standard components of self-distillation and are also used by PSR. SPD introduces only two method-specific costs: (1) a one-time gradient computation and SVD on the calibration set, with cost $(O(N_{\text{cal}})$ and independent of the training-set size; and (2) a $d \times d$ projection applied through the generation hook, with a per-token cost of $O(Ld^2)$, where L is the number of projected layers and d is the hidden dimension. On GSM8K with Qwen2.5-0.5B-Instruct, these additional operations increase the total runtime by only approximately 0.57% compared with PSR.

Table 15: Runtime comparison between PSR and SPD.

Stage	PSR	SPD
Subspace Extraction (one-time)	–	3.5 s
Generation	3186 s	3206 s (+20 s hook overhead)
Fine-tuning	1014 s	1014 s
Total Runtime	**4200** s	**4224** s
Additional Overhead	–	**24** s (**+0.57%**)

B.7 Additional Results on More Backbones

Table 16 summarizes the domain-specific performance and in-domain transfer results for Qwen3.5-4B. SPD achieves the best performance in this new setting, consistent with the main results, further demonstrating strong generalizability across different model families.

Table 16: Domain-specific and in-domain generalization results Qwen3.5-4B.

Backbone	Method	Code		Math		QA	
		MBPP ↑	CodeAlpaca ↓	GSM8K ↑	SVAMP ↑	MMLU ↑	BBH ↑
Qwen3.5-4B	Base model	$76.0_{\pm 0.0}\%$	$0.748_{\pm 0.001}$	$47.0_{\pm 0.0}\%$	$83.0_{\pm 0.0}\%$	$61.0_{\pm 0.5}\%$	$52.2_{\pm 0.5}\%$
	PSR	$74.0_{\pm 1.7}\%$	$0.759_{\pm 0.004}$	$49.0_{\pm 2.6}\%$	$72.0_{\pm 4.0}\%$	$66.5_{\pm 0.5}\%$	$54.7_{\pm 1.1}\%$
	SSD	$72.0_{\pm 2.6}\%$	$0.746_{\pm 0.003}$	$48.0_{\pm 1.7}\%$	$84.0_{\pm 1.0}\%$	$66.0_{\pm 1.7}\%$	$51.7_{\pm 1.6}\%$
	SPD	$\mathbf{79.0}_{\pm \mathbf{2.0}}\%$	$\mathbf{0.722}_{\pm \mathbf{0.001}}$	$\mathbf{51.0}_{\pm \mathbf{2.0}}\%$	$\mathbf{87.0}_{\pm \mathbf{1.7}}\%$	$\mathbf{69.0}_{\pm \mathbf{1.0}}\%$	$\mathbf{59.7}_{\pm \mathbf{0.9}}\%$

C Distillation Paradigms and Self-Policy Distillation

C.1 Preliminaries

We first introduce the notation used in this section. Let $q \sim \mathcal{D}$ denote an input prompt sampled from the prompt distribution $\mathcal{D}$. For a prompt–completion pair (q, y), we use

$$z = T(q, y) = (z_1, \ldots, z_T)$$

to denote the tokenized sequence. A language model policy is denoted by $f_\theta(y \mid q)$, where θ represents the model parameters. Equivalently, the token-level likelihood of z is factorized as

$$p_\theta(z) = \prod_{t=1}^{T} p_\theta(z_t \mid z_{<t}). \tag{15}$$

We use f_T to denote an external teacher model, f_{off} to denote an offline trajectory distribution, and $f_{\theta_{\text{old}}}$ to denote the student model before fine tuning. For curated self-distillation methods, $S(q, y)$ denotes an external scoring or weighting signal, such as correctness filtering, execution feedback, or reward-guided search.

In SPD, we instead define an internal hook transformation

$$\mathcal{T}^{\text{hook}}_{\theta_{\text{old}}}, \tag{16}$$

which induces a self-policy:

$$f^{\text{hook}}_{\theta_{\text{old}}} = \mathcal{T}^{\text{hook}}_{\theta_{\text{old}}}\left(f_{\theta_{\text{old}}}\right). \tag{17}$$

This policy is not an external teacher. It is an internally transformed version of the student's own policy, induced by projection hooks derived from a KV capability subspace.

C.2 Off-Policy Distillation

Off-policy distillation trains the student on trajectories generated by teacher model or collected from a fixed offline dataset:

$$y \sim f_{\text{off}}(\cdot \mid q). \tag{18}$$

Given $z = T(q, y)$, a common token-level distillation objective is

$$\min_\theta \mathbb{E}_{q\sim\mathcal{D},\, y\sim f_{\text{off}}(\cdot|q)} \sum_{t=1}^{T} \mathrm{KL}\left(p_T(\cdot \mid z_{<t}) \,\|\, p_\theta(\cdot \mid z_{<t})\right). \tag{19}$$

Here, the training prefixes $z_{<t}$ are induced by the offline trajectory source. The main limitation is the train–inference mismatch: during training, the student is optimized under prefixes generated by another policy, while at inference time it must condition on its own generated prefixes.

C.3 On-Policy Distillation

On-policy distillation reduces this mismatch by training on trajectories generated by the student itself. At each iteration, the pre-update student samples

$$y \sim f_{\theta_{\text{old}}}(\cdot \mid q). \tag{20}$$

Given $z = T(q, y)$, an external teacher then provides token-level supervision on these student-visited states:

$$\min_{\theta} \mathbb{E}_{q\sim\mathcal{D},\, y\sim f_{\theta_{\text{old}}}(\cdot|q)} \sum_{t=1}^{T} \mathrm{KL}\left(p_T(\cdot \mid z_{<t}) \,\|\, p_\theta(\cdot \mid z_{<t})\right). \tag{21}$$

Here, the training prefixes $z_{<t}$ are generated by the student model itself. Compared with off-policy distillation, on-policy distillation better aligns the training distribution with inference-time behavior. However, it still requires repeated student rollouts and external teacher supervision, making it costly and difficult to scale.

C.4 Self-Distillation

Self-distillation removes the external teacher and trains the model on supervision derived from itself. A basic form samples outputs from the pre-update student:

$$y \sim f_{\theta_{\text{old}}}(\cdot \mid q), \tag{22}$$

and then trains the updated model on these self-generated outputs. Given $z = T(q, y)$, the objective is

$$\min_{\theta} -\mathbb{E}_{q\sim\mathcal{D},\, y\sim f_{\theta_{\text{old}}}(\cdot|q)} \left[\sum_{t=1}^{T} \log p_\theta(z_t \mid z_{<t})\right]. \tag{23}$$

However, this objective directly imitates samples from the model's own unmodified distribution. As a result, it may reinforce existing errors, stylistic artifacts, or task-irrelevant patterns rather than improving the target capability.

To improve the quality of self-generated data, many methods introduce an external scoring or filtering signal:

$$s = S(q, y), \tag{24}$$

leading to a weighted self-distillation objective:

$$\min_{\theta} -\mathbb{E}_{q\sim\mathcal{D},\, y\sim f_{\theta_{\text{old}}}(\cdot|q)} \left[S(q, y) \sum_{t=1}^{T} \log p_\theta(z_t \mid z_{<t})\right], \qquad z = T(q, y). \tag{25}$$

Depending on how trajectories are collected, self-distillation can be implemented in either an off-policy form, where self-generated data are fixed in advance, or an on-policy form, where data are regenerated from the current model during training. However, many effective variants still require external curation signals, such as correctness filtering, execution feedback, or reward-guided search, which introduce additional cost and may be unavailable for the best-performing frontier models.

C.5 Self-Policy Distillation

Self-Policy Distillation (SPD) does not directly train on raw outputs sampled from the original $f_{\theta_{\text{old}}}$. Instead, it first induces a self-policy through an internal transformation:

$$f^{\text{hook}}_{\theta_{\text{old}}} = \mathcal{T}^{\text{hook}}_{\theta_{\text{old}}}\left(f_{\theta_{\text{old}}}\right). \tag{26}$$

Here, $\mathcal{T}^{\text{hook}}_{\theta_{\text{old}}}$ is implemented by the projection hooks defined in Eq. equation 12. These hooks project KV activations onto the selected capability subspace during generation.

Training completions are then sampled from this internally transformed policy:

$$\hat{y} \sim f^{\text{hook}}_{\theta_{\text{old}}}(\cdot \mid q). \tag{27}$$

This gives the self-generated corpus

$$\mathcal{D}_{\text{corpus}} = \{(q^{(i)}, \hat{y}^{(i)})\}_{i=1}^{N_{\text{train}}}. \tag{28}$$

For fine-tuning, each self-generated pair is tokenized as

$$z^{(i)} = T(q^{(i)}, \hat{y}^{(i)}). \tag{29}$$

After removing the hooks, the original model is trained to absorb the behavior of the hooked self-policy:

$$\min_{\Delta\theta_{\text{LoRA}}} - \sum_{(q^{(i)},\hat{y}^{(i)})\in\mathcal{D}_{\text{corpus}}} \sum_{t=2}^{|z^{(i)}|} \log p_{\theta_{\text{old}}+\Delta\theta_{\text{LoRA}}}\left(z^{(i)}_t \mid z^{(i)}_{<t}\right). \tag{30}$$

Equivalently, the procedure can be summarized as follows, with the full procedure detailed in Algorithm 1.

$$f_{\theta_{\text{old}}} \xrightarrow{\mathcal{T}^{\text{hook}}_{\theta_{\text{old}}}} f^{\text{hook}}_{\theta_{\text{old}}} \xrightarrow{\text{sample}} \hat{y} \xrightarrow{\text{distill}} f_\theta. \tag{31}$$

The key difference is that the distillation target is neither provided by an external teacher nor obtained by directly using raw or externally curated self-generated outputs. Instead, it comes from an internally selected self-policy induced from the model's own KV representations.

C.6 Summary

The above paradigms differ along two axes: the source of the trajectory distribution and the source of supervision. Table 17 summarizes the distinction.

Table 17: Comparison of different distillation paradigms.

Method	**Trajectory source**	**Supervision source**
Off-policy distillation	f_{off}	External teacher or offline data
On-policy distillation	$f_{\theta_{\text{old}}}$	External teacher f_T
Self-distillation	$f_{\theta_{\text{old}}}$	curation signal $S(q, y)$
Self-policy distillation	$f^{\text{hook}}_{\theta_{\text{old}}}$	Internally induced self-policy

In short, off-policy and on-policy distillation mainly differ in where the training trajectories come from, while self-distillation differs in where the supervision comes from. Self-policy distillation further refines self-distillation by replacing raw self-generated outputs with samples from an internally induced self-policy.

Algorithm 1 Self-Policy Distillation (SPD)

Require: Student model $f_{\theta_{\text{old}}}$, calibration set $\mathcal{D}_{\text{cal}}$, prompt set $\mathcal{D}_{\text{train}}$, target layers $\mathcal{L}$, rank r
Ensure: Fine-tuned model f_θ

Phase 1: Capability Subspace Extraction

for each target layer $\ell \in \mathcal{L}$ **do**
  for each activation type $A \in \{K, V\}$ **do**
    Compute the correctness-aligned loss $\mathcal{L}_{\text{align}}$ on $\mathcal{D}_{\text{cal}}$
    Collect token-level gradients w.r.t. activation $A^{(\ell)}$
    Aggregate token-level gradients induced by $\mathcal{L}_{\text{align}}$ to form $G_A^{(\ell)}$
    Perform SVD on $G_A^{(\ell)}$
    Select the top-r right singular vectors $V_{A,r}^{(\ell)}$
    Form the projection matrix $P_A^{(\ell)} = V_{A,r}^{(\ell)} \big(V_{A,r}^{(\ell)}\big)^\top$
  end for
end for

Phase 2: Capability-Selective distillation

Apply $P_K^{(\ell)}$ and $P_V^{(\ell)}$ as projection hooks at each $\ell \in \mathcal{L}$
for each prompt $q^{(i)} \in \mathcal{D}_{\text{train}}$ **do**
  Generate one completion with hooks active: $\hat{y}^{(i)} \sim f^{\text{hook}}_{\theta_{\text{old}}}(\cdot \mid q^{(i)})$
end for
Construct the self-generated corpus $\mathcal{D}_{\text{corpus}} = \{(q^{(i)}, \hat{y}^{(i)})\}_{i=1}^{N_{\text{train}}}$
Remove all projection hooks
Fine-tune LoRA parameters on $\mathcal{D}_{\text{corpus}}$ with standard next-token prediction loss
return Fine-tuned model f_θ

D Failure Cases of Projection Hooks

We further analyze a failure case where a code-domain capability subspace is applied to a math problem, leading to negative transfer.

Question: Two trains leave San Rafael at the same time. They travel westward, both for 80 miles. The next day they travel north, covering 150 miles. What's the distance covered by each train in the two days?

Base model:

> To find the distance covered by each train in the two days, we add the distances traveled on each day.
>
> The first train travels 80 miles westward and then 150 miles northward, so it covers $80 + 150 = 230$ miles.
>
> The second train also travels $80 + 150 = 230$ miles.
>
> The answer is 230.

Base model + Hooks:

> To find the distance covered by each train in the two days, we add their distances together.
>
> The first train covers a total of $2 \times 80 = 160$ miles.
>
> The second train covers a total of 150 miles.
>
> Therefore, the answer is $160 + 150 = 310$ miles.

The base model correctly treats the two travel segments, 80 miles westward and 150 miles northward, as distances traveled by *each* train, and therefore computes $80+150 = 230$ miles per train. In contrast, the hooked model incorrectly interprets the two travel segments as if they were assigned across the two trains: it computes $2 \times 80 = 160$ miles for the first train and 150 miles for the second train, and then incorrectly sums them to obtain 310 miles. The key error is therefore a misinterpretation of the problem structure: the hook changes the attribution of the two distance segments from "per train" to "across trains."

In this setting, **the hook is computed from a code-domain calibration set but applied to a math problem, creating a case of cross-domain negative transfer.** During generation, it acts as a filter at two attention layers, retaining only the internal directions identified by the 20 code calibration examples as useful for reaching correct answers in the code domain. **While these directions can improve focus within the code domain, the directions removed by the projection may still contain information that is important for mathematical reasoning.** As a result, the code-domain-specific projection suppresses information needed to correctly represent the structure of the math problem, leading the model to misassign the travel segments and produce an incorrect answer even though the base model answers correctly.

E Further Analysis

We examine representative outputs from the Code, Math, and QA domains to understand how SPD changes the self-generation process. These case studies compare base-model outputs with SPD-generated data and highlight the qualitative effect of projection hooks.

In the **Code** domain, the base model often produces verbose and noisy outputs, mixing the core implementation with extra `print` statements, comments, and long explanations. These explanations can also be misleading. For example, in `remove_Occ`, the code only removes the first occurrence, while the explanation claims that both the first and last occurrences are removed. In contrast, SPD-generated data are more compact and implementation-focused, with less redundant explanation and formatting noise. Overall, SPD-generated data are more task-focused and therefore more suitable for subsequent self-distillation.

In the **Math** domain, the main difference lies in output focus and reasoning structure. The base model may occasionally produce the correct answer, as in *James Runs Sprints*, but it often continues into unrelated follow-up problems, introducing dataset-style continuation noise. It can also misread key conditions, as in *Janet's Ducks*, where it incorrectly introduces weekly calculations such as 3×7 and 4×7. In contrast, SPD-generated data are more structured and answer-oriented. They tend to list intermediate quantities, follow a step-by-step reasoning format, and end with a clear final-answer statement. Thus, SPD-generated data mainly make math outputs cleaner and more focused, rather than directly fixing every reasoning error during generation.

In the **QA** domain, SPD-generated data better preserve the local question type and target reasoning region. In the tongue-tip lymph drainage example, the baseline drifts to a different anatomy topic

about the parotid gland, while SPD-generated data stay closer to tongue-tip lymphatic drainage and related lymph node choices. In the two-proportion z-test example, the baseline continues with a generic test-selection question, whereas SPD-generated data focus on the key condition in the original problem: the number of successes and failures being too small for a valid two-proportion z-test. These examples suggest that SPD-generated data do not merely preserve the multiple-choice format, but steer generation toward more task-relevant QA continuations.

Across the three domains, the main advantage of SPD-generated data lies in its stronger *focus* and *structure*. In the **Code** domain, SPD-generated data reduces verbose explanations and formatting noise, making the outputs more implementation-centered. In the **Math** domain, it encourages more structured and answer-oriented reasoning traces. In the **QA** domain, it keeps continuations closer to the target question type and reasoning region, rather than drifting into generic dataset-style questions. Overall, SPD-generated data does not guarantee single-sample correctness, but provides more targeted and less noisy supervision for subsequent self-distillation.

Code Domain

```
[Code Example 1: remove_Occ]

Base Model:
def remove_Occ(s, ch):
 if s.find(ch) != -1:
 return s[:s.find(ch)] + s[s.find(ch)+1:]
 else:
 return s

print(remove_Occ("hello","l")) # Should print "heo"
print(remove_Occ("abcda","a")) # Should print "bcd"
print(remove_Occ("PHP","P")) # Should print "H"

This solution works by using Python's built-in `find` method to find the index of the
     first occurrence of the target character in the string. If the target character is
     found, it removes the first occurrence by slicing the string starting at that index
     and ending at the next index after the first occurrence. If the target character is
     not found, it simply returns the original string without any modifications. The
     function then slices the remaining part of the string starting from the second
     character (index 1) until the end of the string. This ensures that only the first and
      last occurrences are removed. The provided test cases check whether the function
     behaves as expected for different inputs.

Self-Policy Distillation:
def remove_Occ(s, c):
    # Your code here
    s = s.replace(c, "", 1)
    i = s.rfind(c)
    return s if i == -1 else s[:i] + s[i + 1:]

print(remove_Occ("hello", "l"))


[Code Example 2: find_Volume]

Base Model:
def find_Volume(a,b,c):
 # calculate area of base triangle
 area = (a * b) / 2

 # calculate height of pyramid
 h = c**2 - (area ** 2)

 # calculate volume of pyramid
 V = (area * h) / 3

 return V

# Test cases
print(find_Volume(10,8,6)) # Expected output: 240
print(find_Volume(3,2,2)) # Expected output: 6
print(find_Volume(1,2,1)) # Expected output: 1

Self-Policy Distillation:
def find_Volume(a,b,c):
 return (a*b*c)/2

print(find_Volume(10,8,6)) # Expected output: 240
```

```
print(find_Volume(3,2,2)) # Expected output: 6
print(find_Volume(1,2,1)) # Expected output: 1


[Code Example 3: text_lowercase_underscore]

Base Model:
python
def text_lowercase_underscore(s):
    for i in range(len(s)):
        if s[i].islower() and (s[i-1] == '_' or s[i+1] == '_'):
            return True
    return False

Self-Policy Distillation:
python
def text_lowercase_underscore(s):
 return any(char.islower() for char in s) and all(char.isupper() for char in s)
```

Math Domain

```
[Math Example 1: Janet's Ducks]

Base Model:
Her total earnings from her breakfast is 3 * 7 = 21 dollars.
Her total earnings from baking muffins is 4 * 7 = 28 dollars.
Her total earnings from selling the eggs is 21 + 28 = 49 dollars.
Therefore, the answer is 49.

Self-Policy Distillation:
Janet's ducks lay 16 eggs per day. She eats 3 eggs for breakfast and uses 4 eggs for
     baking muffins.

Remaining eggs to sell:
16 - 3 - 4 = 9

She sells each remaining egg for $2, so her daily earnings are:
9 * 2 = 18

Therefore, Janet makes $18 every day at the farmers' market.

Final answer: 18.


[Math Example 2: James Runs Sprints]

Base Model:
He runs 3 * 3 * 60 = 540 meters per week. The answer is 540.

Question: A certain type of bacteria doubles in quantity every hour. If there
are initially 100 bacteria, how many will be present after 2 hours?

Answer: After the first hour, there will be 100 bacteria x 2 = 200 bacteria.
After the second hour, there will be 200 bacteria x 2 = 400 bacteria.
The answer is 400.

Self-Policy Distillation:
To calculate the total number of meters James runs in a week, you multiply the
distance of one sprint by the number of sprints per day and then by the number
of days per week.

Distance per sprint = 60 meters
Number of sprints per day = 3

Total distance per week = 60 * 3 = 180 meters

Therefore, James runs 180 meters every week.

Final answer: 180 meters.
```

QA Domain

```
[QA Example 1: Tongue Tip Lymph Drainage]

Base Model:
D

Question: Which of the following statements about the lymphatics draining from the parotid
      gland is correct?
A. It drains only to the submental nodes.
B. It drains bilaterally to the submandibular nodes.
C. It drains to the deep cervical nodes.
D. It drains through the facial vein.

Self-Policy Distillation:
C

The following are multiple choice questions about anatomy.

Question: The lymphatic drainage of the tip of the tongue is most closely related to which
      lymph node group?
A. Left submandibular lymph node.
B. Left and right submental lymph nodes.
C. Deep cervical lymph nodes.
D. Parotid lymph nodes.


[QA Example 2: Two-Proportion Z-Test]

Base Model:
C

Question: In a study examining the effectiveness of a new teaching method on student
     performance, which of the following statistical tests should be used?
A. One-sample t-test
B. Paired t-test
C. Two-proportion z-test
D. Chi-square goodness-of-fit test

Self-Policy Distillation:
C

Question: For a two-proportion z-test, why is the test inappropriate when the number of
     successes and failures is too small in one group?
A. The sample sizes are different.
B. The population distributions are unknown.
C. The normal approximation condition is not satisfied.
D. The response variable is continuous.
```