CONTINUAL LEARNING MECHANISMS COMPOSE FOR LONG-HORIZON MEMORIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Language models may need to internalize information that arrives over time and retain it through many subsequent updates. To study this challenge, we introduce long-horizon *memorization*, a setting in which a model learns 100 query-answer tasks through continual supervised fine-tuning without retaining earlier training examples or receiving task identifiers at inference. Sequential updates cause catastrophic forgetting, and no single continual learning mechanism we evaluate maintains strong retention at this horizon. We hypothesize that mechanisms addressing complementary sources of forgetting will be more effective when composed. We organize these compositions along two design dimensions. Data, function, and weight anchors specify what prior information each update should preserve, while low-rank allocation rules determine where successive updates are retained. To test this hypothesis systematically, we construct three distinct 100-task memorization datasets. We introduce task-level successive halving to search the combinatorial design space and use a factorial experiment to measure individual and interaction effects. Our best method combines all three anchors with merged LoRA, ranks among the top 3 methods in all datasets, and raises average final retention from 1.2% under naive sequential fine-tuning to 34.9%, a **28-fold** improvement. The data anchor and merged LoRA provide the largest average gains and interact super-additively on all three datasets. Together, these results show that composing complementary mechanisms substantially improves long-horizon memorization beyond what any individual mechanism achieves.

1 INTRODUCTION

Consider a language model that learns new information over time by updating its parameters. For these parameters to serve as memory, the model must remember what it has learned even after many more updates. We call this setting long-horizon *memorization*. Prompting and retrieval can provide new information at inference time (Brown et al., 2020; Lewis et al., 2020), but the information remains outside the model's parameters and must be supplied again. We instead ask whether repeated updates can build and preserve this memory within the model itself.

We study this problem through continual supervised fine-tuning (SFT) in the domain-incremental setting (Van de Ven & Tolias, 2019). Each task contains a set of query-answer pairs. The model learns 100 tasks in sequence without retaining raw examples from earlier tasks, and it receives no task identifier at inference. The goal is to learn each new task while retaining associations learned from previous tasks. This is difficult because updates for new tasks can overwrite previously stored knowledge, causing catastrophic forgetting (McCloskey & Cohen, 1989; French, 1999).

Prior work shows that mixing rehearsal with knowledge distillation has strong performance (Buzzega et al., 2020), but it does not systematically study the broader space of mechanism compositions. We therefore hypothesize that mechanisms addressing complementary sources of forgetting will retain associations more effectively when composed. To test this hypothesis, we organize compositions along two design dimensions: anchors and low-rank allocation rules. Anchors specify what prior information an update should preserve. We study data, function, and weight anchors, instantiated by generative replay (Shin et al., 2017), self-distillation (Li & Hoiem, 2017), and importance-based regularization (Kirkpatrick et al., 2017; Zenke et al., 2017), respectively. Low-rank allocation rules determine where successive LoRA updates are retained across tasks (Hu et al., 2022). We

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

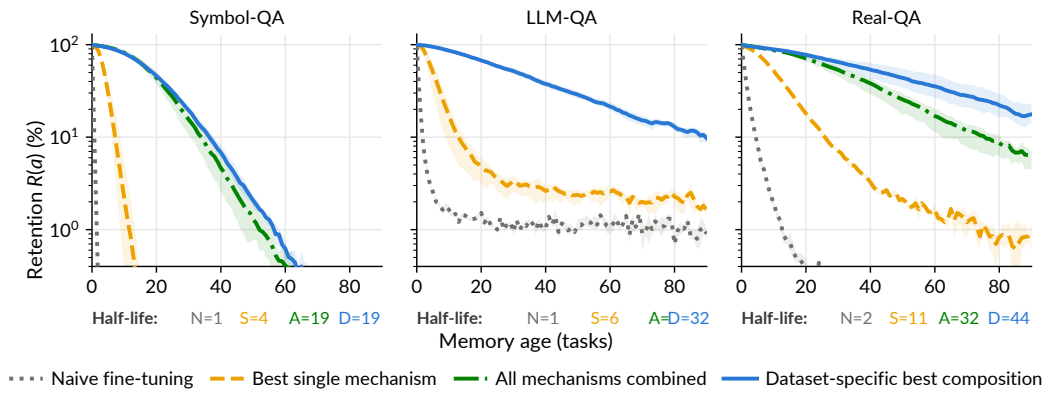

Figure 1: Memory lifetime under the best single mechanism and composed continual learning methods. Lines show three-seed means with min-max bands. Half-life counts tasks until retention halves. Combining multiple anchors with merged LoRA substantially reduces catastrophic forgetting and extends memory lifetime beyond the best single mechanism.

study shared LoRA, which reuses the same adapter across tasks, and merged LoRA, which folds each update into the model before initializing a new adapter.

Testing this hypothesis requires datasets that isolate long-horizon memorization and a method to compare many compositions. Existing public benchmarks for continual language learning primarily measure transfer or performance across heterogeneous downstream tasks or changing corpora (Zhang et al., 2023; Wang et al., 2023b; Jang et al., 2022). Sequential model-editing benchmarks instead study targeted edits rather than task-wise SFT (Hartvigsen et al., 2023; Li & Chu, 2024). To address the lack of an appropriate evaluation pipeline, we construct three datasets, each containing 100 tasks, spanning arbitrary symbol associations, LLM-generated fictional facts, and natural questions filtered from public QA datasets. We introduce task-level successive halving to obtain preliminary evidence across many candidate compositions, then use a factorial experiment to measure individual and interaction effects.

Our experiments support the composition hypothesis. Figure 1 shows that combining multiple anchors with merged LoRA extends memory lifetime beyond individual mechanisms across all three datasets. After 100 tasks, naive sequential fine-tuning achieves 1.2% average final retention, measured as accuracy over all learned tasks after the last update, while the best individual mechanism reaches 8.1%. Our best method combines all three anchors with merged LoRA, which is the only composition in the main factorial that ranks consistently among the top 3 methods in all datasets and achieves 34.9% average final retention, a 28-fold improvement over naive sequential fine-tuning. The factorial analysis identifies the data anchor and merged LoRA as the largest sources of improvement and finds a super-additive interaction between them on all three datasets.

Our contributions are fourfold. First, we formulate long-horizon continual memorization as a distinct continual learning problem for language models. Second, we organize the design space of mechanism compositions around anchors and low-rank allocation rules. Third, we introduce three 100-task memorization datasets, task-level successive halving, and a factorial evaluation of mechanism combinations. Finally, we show that composing all three anchors with merged LoRA substantially improves retention and ranks among the top 3 methods in all datasets.

2 Related Work

Catastrophic interference, often called catastrophic forgetting, was first documented in connectionist neural networks and remains a central problem in continual learning (McCloskey & Cohen, 1989; French, 1999; Kirkpatrick et al., 2017). Contemporary formulations distinguish among task-, domain-, and class-incremental learning according to whether task identity is available at inference time and how the prediction space changes across tasks (Van de Ven & Tolias, 2019). Existing methods broadly rely on regularization, replay, or parameter isolation

(De Lange et al., 2021; Wang et al., 2024). Weight regularization limits changes to parameters that earlier tasks rely on (Kirkpatrick et al., 2017; Schwarz et al., 2018; Zenke et al., 2017; Aljundi et al., 2018), while function regularization preserves earlier model outputs or representations (Li & Hoiem, 2017). Replay uses stored examples (Rebuffi et al., 2017; Rolnick et al., 2019) or generated samples (Shin et al., 2017), whereas parameter isolation assigns different model capacity to different tasks (Rusu et al., 2016; Mallya & Lazebnik, 2017). Some methods combine these signals. Dark Experience Replay jointly uses stored examples and their earlier logits, while Momentum Knowledge Distillation adds a teacher constraint to online continual-learning methods (Buzzega et al., 2020; Michel et al., 2023).

Continual learning methods were largely developed on sequential image classification benchmarks based on MNIST, CIFAR, and ImageNet (LeCun et al., 1998; Krizhevsky & Hinton, 2009; Deng et al., 2009). Later work extends regularization, replay, and benchmarking to sequential language modeling, instruction tuning, and multitask learning (Sun et al., 2019; Scialom et al., 2022; Zhang et al., 2023; Wang et al., 2023b; Xiang et al., 2023). Continual pretraining instead adapts language models as new corpora, domains, or time periods become available (Ke et al., 2023; Ibrahim et al., 2024; Jin et al., 2022; Qin et al., 2022; Jang et al., 2022). Low-Rank Adaptation (LoRA) freezes the pretrained model weights and injects trainable rank decomposition matrices into the model to increase training efficiency while better preserving prior knowledge (Hu et al., 2022). ReLoRA provides a related mechanism for accumulating low-rank updates: during pretraining, it repeatedly merges them into the model and reinitializes the low-rank matrices (Lialin et al., 2024). Other LoRA-based methods aim to reduce interference across tasks. O-LoRA assigns each task a new update subspace and discourages overlap with earlier subspaces during training (Wang et al., 2023a). OSRM instead uses task features to choose update subspaces that reduce interference when merging independently trained task models (Zhang & Zhou, 2025). Sequential model editing addresses a related problem by asking whether a language model can retain many targeted corrections. Existing studies develop explicit storage for successive edits and show that repeated editing can weaken earlier edits and damage other model capabilities (Hartvigsen et al., 2023; Li & Chu, 2024; Gupta et al., 2024). These memory retention challenges also arise for agents interacting with an environment. AgentOdyssey evaluates test-time continual learning agents in procedurally generated text games, with diagnostic tests of world knowledge acquisition and episodic memory (Zhang et al., 2026).

We instead study long-horizon memorization by measuring the recall of associations across one hundred sequential query-answer tasks in the domain-incremental setting, without task identifiers at inference. Rather than introducing another standalone mechanism, we compose generative replay at the data level, self-distillation at the function level, and importance-based regularization at the parameter level. We combine these anchors with merged LoRA and show that the resulting method substantially outperforms each single mechanism.

3 Composing Continual Learning Mechanisms

3.1 Long-Horizon Memorization via Continual SFT

We consider an autoregressive language model p_Θ, parametrized by Θ_0, and a sequence of T supervised tasks that arrive one at a time. Task t contains training data $\mathcal{D}_t$, where each example consists of a query x and its target answer y. When task t arrives, the model has access to $\mathcal{D}_t$ and inherits the state S_{t-1}, including the model parameters Θ_{t-1}. It may not retain or revisit raw training examples from earlier tasks. Let Θ_t denote the model after learning task t. The main challenge of continual supervised fine-tuning (SFT) is to reduce catastrophic forgetting while maintaining the plasticity needed to learn new tasks. The current-task SFT loss is represented by:

$$\mathcal{L}^t_{\mathrm{SFT}}(\Theta) = -\mathbb{E}_{(x,y)\sim\mathcal{D}_t}\left[\log p_\Theta(x,y)\right]. \tag{1}$$

Here $p_\Theta(x,y)$ is the likelihood of the training sequence. Appendix A.2.1 specifies the token-level loss. We do not mask the query tokens because, in practice, it is hard to separate the query from the answer in applications such as test-time training.

Data, function, and weight anchors defined in Section 3.2 add three retention terms to this objective:

$$\Theta_t = \arg\min_\Theta \mathcal{L}^t_{\mathrm{SFT}}(\Theta) + \mathcal{R}^t_D(\Theta) + \mathcal{R}^t_F(\Theta) + \mathcal{R}^t_W(\Theta). \tag{2}$$

Here $\mathcal{R}_D^t$, $\mathcal{R}_F^t$, and $\mathcal{R}_W^t$ correspond to the data, function, and weight anchors. Appendix A.2.2 presents the full objective used to combine the anchors. Low-rank allocation determines which low-rank parameters are updated and what is retained for future tasks. Section 3.3 describes the low-rank allocation rules in detail.

3.2 THREE ANCHORS

Data anchor. A data anchor trains the model on replayed sequences that represent earlier tasks. Let Q_{t-1} be a distribution over these sequences, and let $\ell_D(\Theta, z)$ denote the loss applied to $z \sim Q_{t-1}$. Its general form is

$$\mathcal{R}_D^t(\Theta) = \mathbb{E}_{z\sim Q_{t-1}}\left[\ell_D(\Theta, z)\right]. \tag{3}$$

Vanilla data replay and generative replay construct Q_{t-1} in different ways, while hard and soft replay use different choices of ℓ_D. Deep Generative Replay uses separate generator and solver networks (Shin et al., 2017). LAMOL uses a single language model but treats sampled pseudo-sequences as hard training targets (Sun et al., 2019).

Our data anchor uses a frozen copy of the previous model to generate complete pseudo-sequences from a single task-agnostic replay token. Before each task after the first, we generate 300 sequences and discard empty outputs. During training, each current-task minibatch is paired with one replay minibatch. The replay weight balances the current-task and replay losses, while the generation temperature controls the randomness of replay sampling. We vary both in the task-level successive-halving study described in Section 4.3. Appendix A.2.8 reports the selected values. The frozen model also provides soft next-token targets for the replay sequences, which are used only while learning the current task. Appendix A.2.3 provides the full generation and replay objectives.

Function anchor. A function anchor constrains constrains the update of the current model on current-task inputs by comparing its predictions with a reference distribution. Let μ_t denote the distribution of current-task inputs, let $q_{t-1}(\cdot \mid x)$ denote the reference distribution for input x, and let d measure their difference. Its general form is

$$\mathcal{R}_F^t(\Theta) = \mathbb{E}_{x\sim\mu_t}\left[d(q_{t-1}(\cdot \mid x), p_\Theta(\cdot \mid x))\right]. \tag{4}$$

Our function anchor uses the previous model to define the reference distribution, following Learning without Forgetting (Li & Hoiem, 2017). Although the data anchor also uses soft targets, it applies them to generated replay sequences, whereas the function anchor applies self-distillation (SD) only to current-task data. Appendix A.2.4 gives the full self-distillation objective.

Weight anchor. A weight anchor constrains updates to model parameters according to their estimated importance for previously learned behavior. Let ϑ denote the parameters of Θ tracked by the weight anchor, let $\vartheta_{t-1}^\star$ be its value before task t, and let H_{t-1} encode the accumulated importance. Its general form is

$$\mathcal{R}_W^t(\Theta) = \frac{1}{2}\big(\vartheta - \vartheta_{t-1}^\star\big)^\top H_{t-1}\big(\vartheta - \vartheta_{t-1}^\star\big), \qquad H_{t-1} \succeq 0. \tag{5}$$

EWC applies this quadratic separately for each previous task, using diagonal Fisher information as the importance weights (Kirkpatrick et al., 2017). Online EWC replaces the growing set of task-specific penalties with one penalty centered at the latest parameters using a running Fisher (Schwarz et al., 2018). Synaptic Intelligence (SI) uses the same diagonal quadratic but estimates importance from contributions accumulated along the optimization path (Zenke et al., 2017). Appendix A.2.5 details the estimators and hyperparameter settings.

3.3 LOW-RANK ALLOCATION

Anchors constrain the current update. A low-rank allocation rule determines which low-rank parameters are used for each task and how the learned update is retained for later tasks. For a pretrained matrix W_0, LoRA can be specified as ρBA, where $A \in \mathbb{R}^{r\times d_{\text{in}}}$, $B \in \mathbb{R}^{d_{\text{out}}\times r}$, and $\rho = \alpha_{\text{LoRA}}/r$ (Hu et al., 2022). Let A_t and B_t denote the LoRA matrices optimized during task t, and let a superscript $\star$ denote their values after training on task t. We mainly consider two ways to carry these

matrices across tasks:

$$W_t = \begin{cases} W_0 + \rho B_t A_t, & \text{shared LoRA,} \\ W_{t-1} + \rho B_t A_t, & \text{merged LoRA.} \end{cases} \tag{6}$$

Shared LoRA continues optimizing the same A and B matrices across all tasks, so $B_t A_t$ represents the single complete LoRA adapter after learning tasks 1 through t. Merged LoRA instead assigns each task a new pair of LoRA matrices. After task t, it folds $\rho B_t^\star A_t^\star$ to the dense matrix W_{t-1} and initializes a new pair of LoRA matrices for the next task. This rule adapts ReLoRA's merge and reinitialization pattern to continual learning by merging the LoRA update into the dense weights after each task and initializing new LoRA matrices and a new optimizer for the next task (Lialin et al., 2024). Both methods retain one dense model and one pair of LoRA matrices per adapted weight matrix, so their retained state size remains constant as the number of tasks increases.

The main experiments compare different anchor combinations using shared LoRA and merged LoRA. Appendix A.5.6 reports separate experiments with O-LoRA and sequential OSRM, whose state grows with the number of tasks. Appendix A.2.6 details the update rules and state complexity of all four methods.

4 Evaluation Setup

4.1 Protocol and metrics

We follow the domain-incremental setting of Van de Ven & Tolias (2019) that the model receives one task at a time and is not given the task identity during inference. Each task contains query-answer pairs. Because we study memorization rather than generalization, each task is evaluated on the same examples used for training. Each dataset is an ordered stream of $T = 100$ tasks. After learning task i, we evaluate the model on every task $j \leq i$. Let $x_{j,n}$ and $y_{j,n}$ denote the query and answer for example n in task j, and let N_j be the number of examples in that task. We write $\hat{y}_i(x)$ for the answer produced by the model after learning task i. The temporal accuracy matrix is

$$M_{i,j} = \frac{1}{N_j} \sum_{n=1}^{N_j} \mathbf{1}[\hat{y}_i(x_{j,n}) = y_{j,n}], \qquad 1 \leq j \leq i \leq T. \tag{7}$$

From this matrix, we report final retention, immediate acquisition, and forgetting:

$$\text{Final} = \frac{1}{T} \sum_{j=1}^{T} M_{T,j}, \quad \text{Diag} = \frac{1}{T} \sum_{j=1}^{T} M_{j,j}, \quad \text{Forget} = \frac{1}{T-1} \sum_{j=1}^{T-1} \left(\max_{j \leq i \leq T} M_{i,j} - M_{T,j} \right). \tag{8}$$

Final is the mean accuracy over all tasks after task T. Diag is the mean accuracy on each task immediately after it is learned. Forget is the average drop from each task's best observed accuracy to its final accuracy. It excludes the last task because no later update can cause it to be forgotten.

4.2 Three memorization datasets

Our three datasets increase in semantic realism. Symbol-QA contains 10,000 random key-value associations. LLM-QA contains 10,000 query-answer pairs generated by an LLM across 100 fictional topics. For these synthetically generated datasets, we ensure that each query maps to exactly one target answer across all tasks. Real-QA contains 5,000 natural query-answer pairs from ten public QA datasets, filtered to exclude items the model answers correctly in any of five sampled completions. Each dataset contains 100 tasks, with 100 examples per task for Symbol-QA and LLM-QA, and 50 for Real-QA. Appendix A.3 gives the source list and full construction of each dataset. Appendices A.2.1–A.2.8 provide the method definitions and experimental settings.

4.3 Searching the Combinatorial Design Space

Crossing the three anchor categories in Section 3.2 with the low-rank allocation rules in Section 3.3 yields many possible continual learning methods. We use this design space to seek preliminary evidence for our hypothesis. Training every method on all 100 tasks in each dataset is expensive,

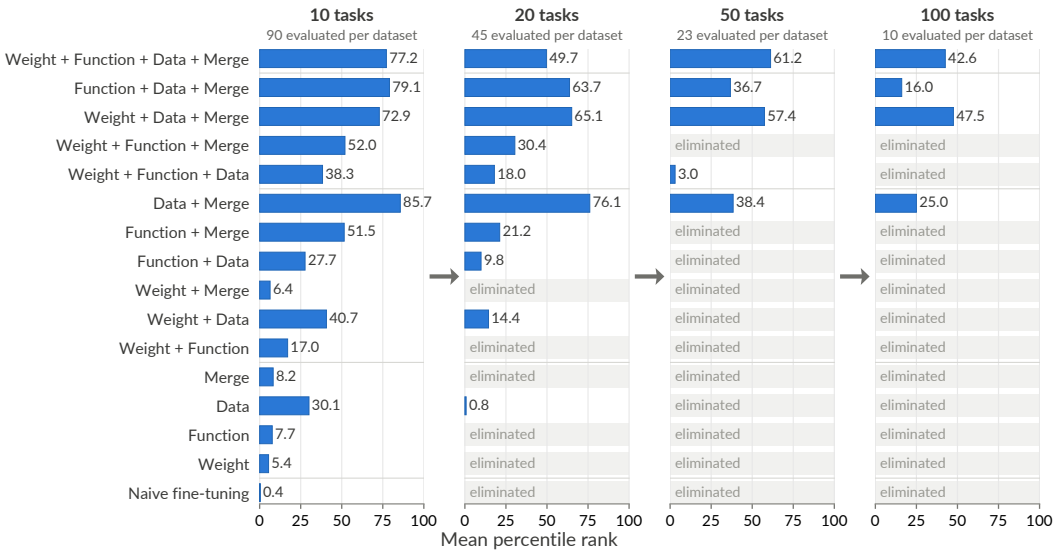

Figure 2: Task-level successive halving. Within each dataset and horizon, we rank surviving candidates by mean retention over three seeds. A method's percentile rank is the percentage of other methods ranked below it. Bars average these values over method variants and then over the three datasets. Because the candidate pool changes, bar heights do not measure changes in retention across horizons. Gray entries indicate that all variants have been eliminated in all three datasets. The leading methods at 100 tasks combine multiple anchors with merged LoRA.

but rankings after only a few tasks may not reliably identify which methods will perform best at longer task horizons. We therefore introduce task-level successive halving (TSH) over progressively longer task horizons.

Task-Level Successive Halving. Unlike prior applications of successive halving that allocate increasing numbers of training iterations to promising hyperparameters (Jamieson & Talwalkar, 2016), our TSH increases the number of sequential tasks and rank configurations by retention at each task horizon. Here, a task horizon r refers to the number of tasks that a configuration of method has learned.

Denote the full search space of method compositions by $\mathcal{A}_1$, and let $\mathcal{S}$ be the set of training seeds. For configuration $a \in \mathcal{A}_1$ and seed $s \in \mathcal{S}$, let $M_{i,j}^{a,s}$ be the resulting temporal accuracy matrix, where $M_{i,j}^{a,s}$ is the accuracy on task j after learning tasks 1 through i. Every seed in $\mathcal{S}$ uses the same task order, fixed by the separate task-order seed, where we call the development task order. Averaging over $\mathcal{S}$ therefore captures training stochasticity but not sensitivity to task order. After r tasks, we score each configuration by its mean final retention across these seeds:

$$F_r(a) = \frac{1}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} \frac{1}{r} \sum_{j=1}^{r} M_{r,j}^{a,s}. \tag{9}$$

Let $\varnothing$ indicate that an anchor is absent. The initial candidate set is

$$\begin{aligned} \mathcal{A}_1 = {} & \{\varnothing, \text{online EWC}, \text{SI}\} \times \{\varnothing, \text{SD}_1, \text{SD}_2\} \\ & \times \{\varnothing, \text{Replay}_1, \ldots, \text{Replay}_4\} \times \{\text{shared LoRA}, \text{merged LoRA}\}, \\ n_1 = {} & |\mathcal{A}_1| = 3 \times 3 \times 5 \times 2 = 90. \end{aligned} \tag{10}$$

SD_1 and SD_2 use self-distillation loss weights 1 and 3. $\text{Replay}_1, \ldots, \text{Replay}_4$ enumerate the Cartesian product of replay loss weights $\{0.5, 0.75\}$ and generation temperatures $\{1.0, 1.5\}$. All other optimization settings remain fixed. Appendix A.2.8 gives the complete settings.

Starting with all 90 configurations, we retain the top 45 after 10 tasks, the top 23 after 20 tasks, and the top 10 after 50 tasks. These final ten configurations continue through all 100 tasks. TSH

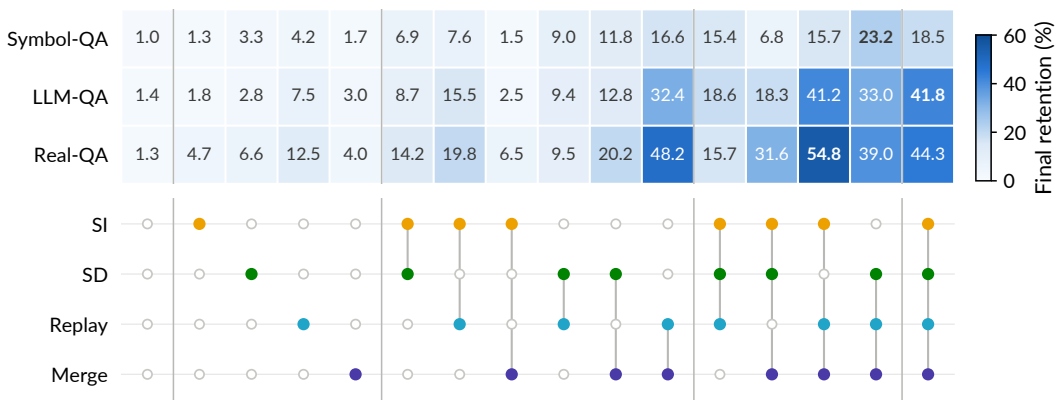

Figure 3: Final retention (%) after 100 tasks for all 16 combinations, averaged over three seeds. Filled markers identify active mechanisms, and an inactive Merge marker indicates shared LoRA. The leftmost column is shared LoRA without anchors. Bold marks the best result per dataset. Appendix A.5.2 reports standard deviations over seeds. The best compositions substantially outperform standalone mechanisms.

therefore uses early retention to decide which configurations receive further training. Although this procedure does not guarantee that it retains the best configuration, the 10-task search rankings show strong agreement with the 100-task final-evaluation rankings for the method compositions evaluated in both phases, despite the different task orders. Algorithm 4 presents the detailed procedure. Appendix A.4.1 provides the resource accounting, and Appendix A.4.5 reports the ranking comparisons.

5 Experiments and Results

Figure 2 shows that no standalone mechanism reaches the 50-task stage of TSH, while every method reaching 100 tasks combines a data anchor with merged LoRA. The winner on each dataset also includes a weight anchor, and the Symbol-QA winner additionally uses a function anchor. These results provide preliminary support for combining multiple anchors with merged LoRA. We therefore evaluate all combinations of the three anchors and merged LoRA in a 2^4 factorial to measure their individual and interaction effects. We use the hyperparameters selected by TSH and the default 100-task order, which differs from the search order. Our primary focus is methods whose retained state remains constant as the number of tasks increases. To test whether state that grows with the number of tasks improves retention, we evaluate O-LoRA and sequential OSRM both individually and as replacements for merged LoRA in each dataset's TSH winner. The full evaluation contains 21 methods per dataset, each with three seeds.

5.1 Compositions outperform standalone mechanisms

Across the full factorial results in Figure 3, compositions are substantially more effective than any standalone mechanism after 100 tasks. The strongest standalone mechanism retains only 4.2% on Symbol-QA, 7.5% on LLM-QA, and 12.5% on Real-QA. By contrast, the highest mean retention among the compositions reaches 23.2%, 41.8%, and 54.8% on the three datasets. The strongest composition also varies by dataset. Symbol-QA favors SD with replay and merged LoRA, LLM-QA favors the full stack of SI, SD, replay, and merged LoRA, and Real-QA favors SI with replay and merged LoRA. Despite this variation, our best method, which combines all three anchors with merged LoRA, is the only composition that ranks among the top 3 methods in all datasets. It achieves 34.9% average final retention across the three datasets. Appendix Table 7 reports the complete cross-dataset ranking. Appendix A.5.2 shows that these methods achieve nearly perfect immediate acquisition, so the differences in final retention primarily reflect forgetting rather than difficulty learning the current task. Appendix A.5.4 also shows how retention changes along a greedy path through the factorial.

	Main effects				Two-way					
Dataset	SI	SD	R	M	SI×SD	SI×R	SI×M	SD×R	SD×M	R×M
SYMBOL-QA	$+0.3$	$\mathbf{+5.7}$	$\mathbf{+9.5}$	$\mathbf{+5.9}$	-0.3	$+0.7$	$\mathbf{-3.0}$	-0.2	$+0.6$	$\mathbf{+3.6}$
LLM-QA	$\mathbf{+5.8}$	$\mathbf{+5.0}$	$\mathbf{+18.5}$	$\mathbf{+14.9}$	$+1.6$	$\mathbf{+2.9}$	-0.1	$\mathbf{-3.5}$	$+1.7$	$\mathbf{+9.4}$
REAL-QA	$\mathbf{+6.3}$	$\mathbf{+3.7}$	$\mathbf{+19.3}$	$\mathbf{+20.5}$	$+1.3$	$+0.1$	$+0.2$	$\mathbf{-10.3}$	$+1.7$	$\mathbf{+11.7}$

	Three-way				Four-way
Dataset	SI×SD×R	SI×SD×M	SI×R×M	SD×R×M	SI×SD×R×M
SYMBOL-QA	$+0.1$	$\mathbf{-1.8}$	-0.8	-1.3	$+0.2$
LLM-QA	-1.3	-0.1	$+0.2$	$\mathbf{-2.6}$	-0.2
REAL-QA	-1.9	$+0.5$	-0.6	$\mathbf{-4.9}$	-0.6

Table 1: Main and interaction effects from the 2^4 factorial analysis of final retention, in percentage points. SI is Synaptic Intelligence (weight anchor), SD is self-distillation (function anchor), R is replay (data anchor), and M is merged LoRA (low-rank allocation rule). Bold marks statistically significant entries ($p < 0.05$). Replay and merged LoRA have the largest main effects and a positive interaction in all three datasets.

Replacing merged LoRA with O-LoRA changes final retention only slightly, with small gains on the two natural-language datasets and a decrease on SYMBOL-QA. Sequential OSRM lowers retention on all three datasets. Thus, low-rank allocation rules with task-growing state do not consistently improve retention. Appendix A.5.6 reports this comparison in detail. We evaluate general capability on held-out mathematical reasoning and knowledge benchmarks: GSM8K (Cobbe et al., 2021), MATH (Hendrycks et al., 2021), MGSM (Shi et al., 2022), and MMLU-Redux (Gema et al., 2025). Improved memorization does not necessarily preserve general capability. All methods exhibit catastrophic forgetting on these benchmarks. Figure 4 shows that O-LoRA preserves substantially more general capability than merged LoRA after training on LLM-QA and REAL-QA, despite only small gains in final retention. Appendix A.5.9 reports the full comparison.

5.2 COMPOSITION EXTENDS MEMORY HALF-LIFE

Final retention summarizes only performance after the last task. To examine how memories decay during training, we group the temporal accuracy matrix by memory age. For age a, $R(a)$ averages all entries $M_{i,j}$ with $i-j = a$, and therefore measures accuracy after a later tasks have been learned. We define memory half-life as the first age at which $R(a)$ falls below half of $R(0)$. Figure 1 compares these trajectories for naive fine-tuning, the strongest standalone mechanism, our best method, and the strongest composition for each dataset.

Naive fine-tuning has a memory half-life of only one task on SYMBOL-QA and LLM-QA, and two tasks on REAL-QA. The strongest standalone mechanism extends these half-lives to 4, 6, and 11 tasks. The strongest composition extends them further to 19, 32, and 44 tasks. Our best method reaches 19, 32, and 32 tasks, matching the strongest composition on the first two datasets and remaining substantially stronger than any standalone mechanism on REAL-QA. Even so, all curves continue to decline with memory age. Composition therefore delays forgetting by changing its timescale, but it does not prevent the eventual loss of older memories.

5.3 REPLAY AND MERGED LORA PROVIDE THE LARGEST GAINS

Table 1 reports the main effects of the four mechanisms and all interactions among them on final retention. Appendix A.5.3 provides the full test statistics.

Replay and merged LoRA have the two largest main effects on every dataset. These main effects measure the average change in final retention from adding a mechanism across all configurations of the other three mechanisms. Replay raises retention by 9.5 to 19.3 percentage points, while merged LoRA raises it by 5.9 to 20.5 points. Their combination is also strongly super-additive.

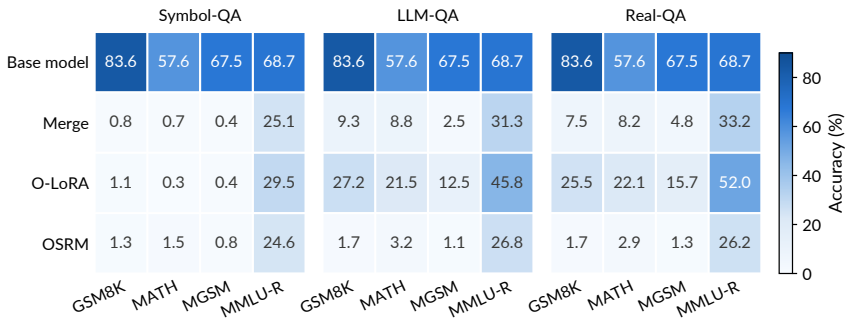

Figure 4: General capability after 100 tasks, averaged over three seeds. Anchors and hyperparameters from TSH are fixed across allocation rules: all three anchors on SYMBOL-QA, and data and weight anchors on LLM-QA and REAL-QA. O-LoRA better preserves general capability on the two natural-language datasets.

The interaction between replay and merged LoRA is positive and significant on all three datasets. Figure 3 shows this synergy directly in the configurations without SI or SD. Across SYMBOL-QA, LLM-QA, and REAL-QA, the standalone gains from replay and merged LoRA sum to only 3.9, 7.7, and 13.9 points, respectively. Combining them instead improves retention over naive fine-tuning by 15.6, 31.0, and 46.9 points. The gains from the pair therefore far exceed the sum of their standalone contributions.

The contributions of SI and SD depend more strongly on the dataset and the other composition. SD has a positive main effect on every dataset, but its negative interaction with replay on LLM-QA and REAL-QA shows that its average benefit is smaller when replay is already present. SI has positive main effects on the two natural-language datasets but no detectable main effect on SYMBOL-QA. Its interaction with merged LoRA is negative and significant only on SYMBOL-QA. As discussed in Appendix A.2.5, merged LoRA replaces the LoRA factors after each task, while SI carries forward importance values and references tied to the previous factors. These values are therefore applied to newly initialized coordinates whose functional roles have changed. The negative interaction on SYMBOL-QA is consistent with this mismatch. Its absence on the natural-language datasets shows that the mismatch creates a structural risk rather than a universal empirical penalty. One possible explanation is that the arbitrary mappings in SYMBOL-QA provide less reusable structure across tasks, making misplaced constraints on the fresh workspace more costly.

6 CONCLUSION

We formalize long-horizon continual memorization and organize methods by data, function, weight anchors, and low-rank allocation rules. We introduce three 100-task query-answer datasets of increasing naturalness, task-level successive halving to search compositions, and a factorial design to measure individual and interaction effects. No individual mechanism retains knowledge well after 100 tasks. Combining all three anchors with merged LoRA yields the only composition ranked in the top 3 on every dataset, raising average final retention from 1.2% under naive fine-tuning to 34.9%, a 28-fold improvement. The data anchor and merged LoRA provide the largest gains and interact super-additively on every dataset. These results support composing complementary mechanisms for long-horizon memorization.

Memorization rather than generalization. Our evaluation tests recall using the queries seen during training. A model may retain the corresponding associations yet answer a paraphrased query incorrectly. Our results therefore do not establish generalization to new query formulations.

General capability preservation. Stronger memorization does not ensure preservation of general capability. The evaluated methods still lose substantial accuracy on general capability benchmarks after 100 tasks (Appendix A.5.9). Preserving these abilities while learning new associations remains an open challenge.

AI USE STATEMENT

In this work, we used generative AI tools for searching related work, implementation, generating synthetic datasets, and polishing writing. We have not used generative AI tools for ideation or for developing the conceptual framework, and formulating mathematical claims and writing proofs are not applicable to this work. Additionally, we used generative AI tools for creating or modifying scientific figures and summarizing existing literature. We have reviewed all AI-assisted work. We have carefully checked the LLM-generated code for correctness and polishing changes. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

Our experiments use synthetic data and public question-answering benchmarks. Applying these methods to sensitive information could create privacy risks through persistent memorization. We also observe substantial loss of general capabilities after continual training. Improved retention alone should therefore not be interpreted as evidence of safe or reliable deployment.

REPRODUCIBILITY STATEMENT

Section 4 specifies the evaluation protocol and metrics. Appendix A.2 documents method implementations and hyperparameters, Appendix A.3 describes dataset sources and construction, and Appendix A.4 specifies the search procedure, seeds, and task orders. Appendix A.5 reports complete results and variation across seeds.

REFERENCES

Rahaf Aljundi, Francesca Babiloni, Mohamed Elhoseiny, Marcus Rohrbach, and Tinne Tuytelaars. Memory aware synapses: Learning what (not) to forget. In *European conference on computer vision*, pp. 144–161. Springer, 2018.

Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. Semantic parsing on freebase from question-answer pairs. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pp. 1533–1544, 2013.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Pietro Buzzega, Matteo Boschini, Angelo Porrello, Davide Abati, and Simone Calderara. Dark experience for general continual learning: a strong, simple baseline. *Advances in neural information processing systems*, 33:15920–15930, 2020.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Matthias De Lange, Rahaf Aljundi, Marc Masana, Sarah Parisot, Xu Jia, Aleš Leonardis, Gregory Slabaugh, and Tinne Tuytelaars. A continual learning survey: Defying forgetting in classification tasks. *IEEE transactions on pattern analysis and machine intelligence*, 44(7):3366–3385, 2021.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255. Ieee, 2009.

Robert M French. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4):128–135, 1999.

Aryo Pradipta Gema, Joshua Ong Jun Leang, Giwon Hong, Alessio Devoto, Alberto Carlo Maria Mancino, Rohit Saxena, Xuanli He, Yu Zhao, Xiaotang Du, Mohammad Reza Ghasemi Madani, et al. Are we done with mmlu? 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. 5069–5096, 2025.

Akshat Gupta, Anurag Rao, and Gopala Anumanchipalli. Model editing at scale leads to gradual and catastrophic forgetting. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 15202–15232, 2024.

Tom Hartvigsen, Swami Sankaranarayanan, Hamid Palangi, Yoon Kim, and Marzyeh Ghassemi. Aging with grace: Lifelong model editing with discrete key-value adaptors. *Advances in neural information processing systems*, 36:47934–47959, 2023.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

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

Adam Ibrahim, Benjamin Thérien, Kshitij Gupta, Mats L Richter, Quentin Anthony, Timothée Lesort, Eugene Belilovsky, and Irina Rish. Simple and scalable strategies to continually pre-train large language models. *arXiv preprint arXiv:2403.08763*, 2024.

Kevin Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *Artificial intelligence and statistics*, pp. 240–248. PMLR, 2016.

Joel Jang, Seonghyeon Ye, Changho Lee, Sohee Yang, Joongbo Shin, Janghoon Han, Gyeonghun Kim, and Minjoon Seo. Temporalwiki: A lifelong benchmark for training and evaluating ever-evolving language models. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 6237–6250, 2022.

Xisen Jin, Dejiao Zhang, Henghui Zhu, Wei Xiao, Shang-Wen Li, Xiaokai Wei, Andrew Arnold, and Xiang Ren. Lifelong pretraining: Continually adapting language models to emerging corpora. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 4764–4780, 2022.

Mandar Joshi, Eunsol Choi, Daniel S Weld, and Luke Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1601–1611, 2017.

Zixuan Ke, Yijia Shao, Haowei Lin, Tatsuya Konishi, Gyuhak Kim, and Bing Liu. Continual pre-training of language models. *arXiv preprint arXiv:2302.03241*, 2023.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

Yann LeCun, Corinna Cortes, and Christopher J. C. Burges. The mnist database of handwritten digits, 1998.

Kenton Lee, Ming-Wei Chang, and Kristina Toutanova. Latent retrieval for weakly supervised open domain question answering. In *Proceedings of the 57th annual meeting of the association for computational linguistics*, pp. 6086–6096, 2019.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020.

Qi Li and Xiaowen Chu. Can we continually edit language models? on the knowledge attenuation in sequential model editing. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 5438–5455, 2024.

Zhizhong Li and Derek Hoiem. Learning without forgetting. *IEEE transactions on pattern analysis and machine intelligence*, 40(12):2935–2947, 2017.

Vladislav Lialin, Sherin Muckatira, Namrata Shivagunde, and Anna Rumshisky. Relora: High-rank training through low-rank updates. In *International Conference on Learning Representations*, volume 2024, pp. 49405–49421, 2024.

Yan-Shuo Liang and Wu-Jun Li. Inflora: Interference-free low-rank adaptation for continual learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 23638–23647, 2024.

Alex Mallen, Akari Asai, Victor Zhong, Rajarshi Das, Daniel Khashabi, and Hannaneh Hajishirzi. When not to trust language models: Investigating effectiveness of parametric and non-parametric memories. In *Proceedings of the 61st annual meeting of the association for computational linguistics (volume 1: Long papers)*, pp. 9802–9822, 2023.

Arun Mallya and Svetlana Lazebnik. Packnet: Adding multiple tasks to a single network by iterative pruning. *arXiv preprint arXiv:1711.05769*, 2017.

Michael McCloskey and Neal J Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, volume 24, pp. 109–165. Elsevier, 1989.

Nicolas Michel, Maorong Wang, Ling Xiao, and Toshihiko Yamasaki. Rethinking momentum knowledge distillation in online continual learning. *arXiv preprint arXiv:2309.02870*, 2023.

Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. Can a suit of armor conduct electricity? a new dataset for open book question answering. In *Proceedings of the 2018 conference on empirical methods in natural language processing*, pp. 2381–2391, 2018.

Ankit Pal, Logesh Kumar Umapathi, and Malaikannan Sankarasubbu. Medmcqa: A large-scale multi-subject multi-choice dataset for medical domain question answering. In *Conference on health, inference, and learning*, pp. 248–260. PMLR, 2022.

Yujia Qin, Jiajie Zhang, Yankai Lin, Zhiyuan Liu, Peng Li, Maosong Sun, and Jie Zhou. Elle: Efficient lifelong pre-training for emerging data. In *Findings of the Association for Computational Linguistics: ACL 2022*, pp. 2789–2810, 2022.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 conference on empirical methods in natural language processing*, pp. 2383–2392, 2016.

Anastasia Razdaibiedina, Ashish Khetan, Zohar Karnin, Daniel Khashabi, and Vivek Madan. Representation projection invariance mitigates representation collapse. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 14638–14664, 2023.

Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H. Lampert. icarl: Incremental classifier and representation learning. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pp. 2001–2010, 2017.

David Rolnick, Arun Ahuja, Jonathan Schwarz, Timothy Lillicrap, and Gregory Wayne. Experience replay for continual learning. *Advances in neural information processing systems*, 32, 2019.

Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016.

Jonathan Schwarz, Wojciech Czarnecki, Jelena Luketina, Agnieszka Grabska-Barwinska, Yee Whye Teh, Razvan Pascanu, and Raia Hadsell. Progress & compress: A scalable framework for continual learning. In *International conference on machine learning*, pp. 4528–4537. PMLR, 2018.

Thomas Scialom, Tuhin Chakrabarty, and Smaranda Muresan. Fine-tuned language models are continual learners. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 6107–6122, 2022.

Freda Shi, Mirac Suzgun, Markus Freitag, Xuezhi Wang, Suraj Srivats, Soroush Vosoughi, Hyung Won Chung, Yi Tay, Sebastian Ruder, Denny Zhou, et al. Language models are multilingual chain-of-thought reasoners. *arXiv preprint arXiv:2210.03057*, 2022.

Hanul Shin, Jung Kwon Lee, Jaehong Kim, and Jiwon Kim. Continual learning with deep generative replay. *Advances in neural information processing systems*, 30, 2017.

Fan-Keng Sun, Cheng-Hao Ho, and Hung-Yi Lee. Lamol: Language modeling for lifelong language learning. *arXiv preprint arXiv:1909.03329*, 2019.

Gido M Van de Ven and Andreas S Tolias. Three scenarios for continual learning. *arXiv preprint arXiv:1904.07734*, 2019.

Liyuan Wang, Xingxing Zhang, Hang Su, and Jun Zhu. A comprehensive survey of continual learning: Theory, method and application. *IEEE transactions on pattern analysis and machine intelligence*, 46(8):5362–5383, 2024.

Xiao Wang, Tianze Chen, Qiming Ge, Han Xia, Rong Bao, Rui Zheng, Qi Zhang, Tao Gui, and Xuan-Jing Huang. Orthogonal subspace learning for language model continual learning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 10658–10671, 2023a.

Xiao Wang, Yuansen Zhang, Tianze Chen, Songyang Gao, Senjie Jin, Xianjun Yang, Zhiheng Xi, Rui Zheng, Yicheng Zou, Tao Gui, Qi Zhang, and Xuanjing Huang. Trace: A comprehensive benchmark for continual learning in large language models. *arXiv preprint arXiv:2310.06762*, 2023b.

Johannes Welbl, Nelson F Liu, and Matt Gardner. Crowdsourcing multiple choice science questions. In *Proceedings of the 3rd Workshop on Noisy User-generated Text*, pp. 94–106, 2017.

Martin Wistuba, Prabhu Teja Sivaprasad, Lukas Balles, and Giovanni Zappella. Continual learning with low rank adaptation. *arXiv preprint arXiv:2311.17601*, 2023.

Jiannan Xiang, Tianhua Tao, Yi Gu, Tianmin Shu, Zirui Wang, Zichao Yang, and Zhiting Hu. Language models meet world models: Embodied experiences enhance language models. *Advances in neural information processing systems*, 36:75392–75412, 2023.

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

Friedemann Zenke, Ben Poole, and Surya Ganguli. Continual learning through synaptic intelligence. In *International conference on machine learning*, pp. 3987–3995. Pmlr, 2017.

Haobo Zhang and Jiayu Zhou. Unraveling lora interference: Orthogonal subspaces for robust model merging. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 26459–26472, 2025.

Zheyuan Zhang, Zehao Wen, Alvin Zhang, Andrew Wang, Jianwen Xie, Daniel Khashabi, and Tianmin Shu. Agentodyssey: Open-ended long-horizon text game generation for test-time continual learning agents. *arXiv preprint arXiv:2606.24893*, 2026.

Zihan Zhang, Meng Fang, Ling Chen, and Mohammad-Reza Namazi-Rad. Citb: A benchmark for continual instruction tuning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 9443–9455, 2023.

A Appendix

A.1 Extended Related Work

Continual-learning settings. Continual learning trains one model on a sequence of tasks while seeking to preserve performance on earlier data. Standard formulations distinguish task-, class-, and domain-incremental learning according to how the output space changes and whether inference provides a task identifier (Van de Ven & Tolias, 2019). Our setting uses the same question–answer interface and vocabulary throughout training and provides no task identifier at evaluation. It therefore tests whether one model can retain all associations without routing each input to a task-specific predictor.

Mechanisms for preserving earlier knowledge. Regularization methods constrain learning through information retained from previous tasks. Weight-based methods assign importance to individual parameters and penalize changes to important values, as in EWC, online EWC, SI, and MAS (Kirkpatrick et al., 2017; Schwarz et al., 2018; Zenke et al., 2017; Aljundi et al., 2018). Function-based methods instead constrain model behavior. Learning without Forgetting, for example, asks the current model to match a previous model's outputs on available inputs (Li & Hoiem, 2017). Replay methods train on earlier examples stored in memory (Rebuffi et al., 2017; Rolnick et al., 2019) or on samples generated to approximate earlier data (Shin et al., 2017). Parameter-isolation methods reduce interference by assigning different parameters to different tasks or by restricting which parameters each task may change (Rusu et al., 2016; Mallya & Lazebnik, 2017). Our data, function, and weight anchors instantiate the first three preservation signals, while low-rank allocation rules determine whether successive task updates reuse or separate low-rank capacity.

Combining continual-learning mechanisms. Prior work combines preservation signals, but it does not systematically study the broader space of mechanism compositions considered here. Dark Experience Replay stores examples together with the logits produced when those examples entered memory, then uses both rehearsal and output matching during later learning (Buzzega et al., 2020). Momentum Knowledge Distillation adds a slowly updated teacher to online continual learning methods and studies how distillation complements replay (Michel et al., 2023). Our study also examines interactions among mechanisms, but it uses generated replay rather than stored examples and evaluates data, function, and weight anchors both individually and in controlled compositions with different low-rank allocation rules.

Continual learning for language models. Prior language work studies sequential language modeling, task learning, and instruction tuning. LAMOL learns to answer current-task examples and generate pseudo-examples for earlier tasks (Sun et al., 2019). Later work examines continual fine-tuning, instruction-tuning benchmarks, and forgetting across diverse language tasks (Scialom et al., 2022; Zhang et al., 2023; Wang et al., 2023b; Xiang et al., 2023). Continual pretraining instead updates a language model as new corpora, domains, or time periods arrive (Jin et al., 2022; Qin et al., 2022; Jang et al., 2022; Ke et al., 2023; Ibrahim et al., 2024). These studies primarily measure language modeling, transfer, or downstream task performance. We isolate associative retention by evaluating the same question–answer items before and after many later tasks.

REPINA mitigates representation collapse during fine-tuning by matching fine-tuned representations to pretrained representations, either directly or through a learned projection (Razdaibiedina et al., 2023). Our function anchor instead matches the previous model's output distributions on current-task data.

Sequential model editing. Sequential model editing also asks a model to retain many updates. GRACE leaves the original model weights unchanged and stores edits in a discrete key–value codebook that activates for inputs near a stored key (Hartvigsen et al., 2023). Other studies show that applying model editors repeatedly can weaken earlier edits, reduce the model's ability to learn new edits, and harm downstream performance (Li & Chu, 2024; Gupta et al., 2024). This literature emphasizes edit success, generalization to related inputs, and limited changes to unrelated behavior. Our setting instead presents sets of associations as ordinary training data and measures recall across one hundred task boundaries. The two settings share the problem of retaining many factual updates, but they differ in how they introduce and evaluate those updates.

Low-rank allocation. LoRA represents an update to a frozen weight matrix with two low-rank matrices that can be merged into the dense weight for evaluation (Hu et al., 2022). ReLoRA repeatedly merges and reinitializes low-rank matrices during pretraining, allowing multiple low-rank updates to produce a higher-rank cumulative change (Lialin et al., 2024). Continual learning methods use this structure in several ways. CoLoR trains a separate LoRA expert for each task and infers which expert to use at evaluation (Wistuba et al., 2023). O-LoRA retains the matrices learned for earlier tasks and penalizes overlap between the current and earlier A matrices (Wang et al., 2023a). InfLoRA constructs task-specific update subspaces to reduce interference with earlier tasks (Liang & Li, 2024). OSRM uses task features to initialize LoRA subspaces before independently training and later merging task models (Zhang & Zhou, 2025). Our comparison separates the low-rank allocation rule from the preservation objective. Shared LoRA reuses one pair of matrices, merged LoRA commits each task's update to the dense weights before creating a new pair, O-LoRA retains a new pair for each task, and our sequential OSRM adaptation combines merged LoRA with an initialization derived only from completed tasks.

A.2 Detailed Method Implementations

A.2.1 Scope, taxonomy, and notation

This appendix specifies the methods in our evaluation suite. We distinguish anchors, which preserve evidence about earlier tasks, from low-rank allocation, which specifies the low-rank parameters used for each task and how learned updates are retained. Vanilla sequential SFT is the no-anchor baseline and uses shared LoRA as its low-rank allocation rule. The following subsections describe each anchor and low-rank allocation rule. The data anchor is unconditional generative replay A.2.3, the function anchor is self-distillation A.2.4, the weight anchors are Synaptic Intelligence (SI) A.2.5, and the online EWC A.2.5. The low-rank allocation rules are shared LoRA A.2.6, merged LoRA A.2.6, O-LoRA A.2.6, and our strictly sequential adaptation of OSRM A.2.6. Table 2 summarizes this categorization.

Category	Method	Persistent learner state	Horizon dependence
Vanilla baseline	Sequential SFT (no anchor)	one shared LoRA workspace	Constant
Data anchor	Unconditional generative replay	Model trained from last task to generate replay data unconditionally. No raw examples or replay samples are carried to the next task	Constant
Function anchor	self-distillation	Model trained from the last task treated as a frozen teacher during the current task for KL estimation	Constant
Weight anchor	Synaptic Intelligence	One reference and one diagonal importance tensor per trainable tensor saved after each task and then discarded before training the next task	Constant
Weight anchor	Online EWC	One reference and one running diagonal Fisher tensor per trainable tensor saved after each task and then discarded before training the next task	Constant
Low-rank allocation	shared LoRA	One continually updated rank-r adapter used by vanilla SFT and composable with any anchor.	Constant
Low-rank allocation	merged LoRA	Trained model from last task only and a fresh rank-r LoRA for the current task	Constant
Low-rank allocation	O-LoRA	Frozen rank-r factors for every completed task and one fresh rank-r LoRA	Linear in tasks
Low-rank allocation	Sequential OSRM	Trained model from last task only, a fresh rank-r LoRA for the current task, and one feature vector per past task and target module	Linear in tasks

Table 2: Methods used in our project, categorized as anchors or low-rank allocation rules. "Constant" refers to a method that depends only on a fixed set of information, while "Linear in tasks" means the method needs to store information that grows with the number of tasks during training.

For task t, let Θ denote the student model being optimized and let $\bar{\Theta}_{t-1}$ denote the frozen model obtained after task $t-1$. A training example is a token sequence $z = (z_1, \ldots, z_L)$ containing the question, the literal answer delimiter, and the boxed answer. When the replay seed is enabled, it is prepended to the sequence. We train on the entire sequence of question and answer pairs since during the deployment phase, we don't usually have the question prefix that we can mask from the SFT loss computation. We mask the replay seed token. For a current-task minibatch $\mathcal{B}$, let L_n denote the number of tokens for minibatch n, our implementation computes

$$\widehat{\mathcal{L}}^{t}_{\mathrm{SFT}}(\Theta, \mathcal{B}) = -\frac{1}{\sum_{n\in\mathcal{B}}\sum_{\ell=1}^{L_n} m_{n,\ell}} \sum_{n\in\mathcal{B}}\sum_{\ell=1}^{L_n} m_{n,\ell} \log p_{\Theta}(z_{n,\ell} \mid z_{n,<\ell}), \tag{11}$$

where $m_{n,\ell} = 1$ at non-padding positions and 0 otherwise. Equation 1 is the expectation of this minibatch loss under data shuffling, so the current-task objective covers the complete sequence. The data-anchor loss instead uses teacher-generated unconditional generative replay sequences and is defined in Section A.2.3.

All low-rank allocation rules operate on the same set $\mathcal{J}$ of adapted linear maps. For $j \in \mathcal{J}$, let W_j^{eff} represent the updated LLM layer j parameters with a LoRA module (Hu et al., 2022), α_{LoRA} and r represent the LoRA α and LoRA ranks, respectively. We can express it as the following:

$$W_j^{\text{eff}} = W_j + \rho B_j A_j, \qquad A_j \in \mathbb{R}^{r \times d_{\text{in},j}}, \quad B_j \in \mathbb{R}^{d_{\text{out},j} \times r}, \quad \rho = \frac{\alpha_{\text{LoRA}}}{r}. \tag{12}$$

During training, dropout is applied to the input of the low-rank branch and is disabled at evaluation. We use Kaiming initialization for A_j and initialize $B_j = 0$, so a fresh adapter initially leaves the model function unchanged. Section A.2.6 describes how each low-rank allocation rule carries these factors across task boundaries.

A.2.2 Full compositional objective

This section states the loss that the optimizer minimizes at task t. A configuration selects one low-rank allocation rule and may add anchors. This rule determines which LoRA parameters remain trainable and how the method carries them across task boundaries. Shared LoRA, merged LoRA, and sequential OSRM do not add terms to the loss. O-LoRA also regularizes its factors during training, so its two regularizers appear in the complete objective below. The following subsections give the mathematical details of every anchor and low-rank allocation rule.

We use binary indicators a_D^t, a_F^t, a_{SI}^t, and a_{EWC}^t for the data, function, SI, and online EWC anchors. An anchor indicator equals one only when the configuration uses that anchor and the anchor has state from an earlier task. All four anchor indicators therefore equal zero at $t = 1$. The indicator a_{O}^t equals one when the configuration uses O-LoRA. Each λ is a nonnegative coefficient that controls the strength of its term.

The data anchor changes the data used to fit the model. Let $\mathcal{L}_D^t$ denote the replay loss on sequences that the model from task $t-1$ generates. When the configuration uses replay and an earlier model exists, we mix this loss with the current task SFT loss instead of treating replay as a separate regularizer:

$$\mathcal{L}_{\text{fit}}^t = \begin{cases} \mathcal{L}_{\text{SFT}}^t, & a_D^t = 0, \\ (1-w)\mathcal{L}_{\text{SFT}}^t + w\mathcal{L}_D^t, & a_D^t = 1. \end{cases} \tag{13}$$

Here $w \in [0, 1]$ sets the replay ratio. Equation 13 reduces to ordinary SFT when the data anchor is off or when $t = 1$.

The other terms serve different roles. The function anchor loss $\mathcal{L}_F^t$ matches the previous model's distribution over the next token on current task sequences. The SI penalty $\mathcal{R}_{\text{SI}}^t$ limits changes to trainable scalar coordinates that contributed strongly along earlier optimization paths. The online EWC penalty $\mathcal{R}_{\text{EWC}}^t$ limits changes to coordinates with a running diagonal Fisher value. For O-LoRA, $\mathcal{R}_{\perp}^t$ discourages overlap between the new input side factors and the factors from earlier tasks, while $\mathcal{R}_2^t$ controls the norm of the current factors. With these terms, the optimizer minimizes

$$\begin{aligned} \mathcal{L}_{\text{impl}}^t = \mathcal{L}_{\text{fit}}^t &+ a_F^t \lambda_F \mathcal{L}_F^t + a_{\text{SI}}^t \lambda_{\text{SI}} \mathcal{R}_{\text{SI}}^t + a_{\text{EWC}}^t \lambda_{\text{EWC}} \mathcal{R}_{\text{EWC}}^t \\ &+ a_{\text{O}}^t \left(\lambda_{\perp} \mathcal{R}_{\perp}^t + \lambda_2 \mathcal{R}_2^t \right). \end{aligned} \tag{14}$$

When the data anchor is active, we pair each current minibatch with one replay minibatch. We create a shuffled replay iterator at the start of each epoch and create another one whenever it is exhausted, so every current minibatch receives a replay partner. Self-distillation is evaluated only on the current minibatch and constrains the current model to match the previous model on current-task sequences. For all low-rank allocation rules except O-LoRA, SI records the gradient of $\mathcal{L}_{\text{fit}}^t$ before adding the function and weight penalties. With O-LoRA, SI records only the weighted current-task loss. In every case, the optimizer updates the model using the complete objective in Equation 14.

A.2.3 Data anchor: unconditional generative replay

The data anchor approximates the distribution of earlier tasks using sequences sampled from the previous model. It stores no raw examples from earlier tasks. At the start of each task, the previous

model generates a temporary replay set that is used only while learning the current task. Generated samples may be retained as diagnostic logs, but they are not part of the learner state and are never used to train later tasks.

Seed token. Let $\mathcal{D}_t$ denote the training data for task t. We add one token, $s =$ `<|replay_token|>`, to the vocabulary and prepend it to every training sequence. For a query-answer pair $(q, a) \in \mathcal{D}_t$, the formatted sequence is

$$z(q, a) = s \,\|\, \texttt{Question: q} \,\|\, \texttt{\textbackslash nAnswer: \textbackslash boxed\{a\}}, \tag{15}$$

where $\|$ denotes concatenation. Because the same token prefixes the training sequences from every task, the model learns to generate complete formatted sequences from $p_\Theta(\cdot \mid s)$. The data anchor samples this distribution instead of storing training data from earlier tasks.

We initialize the input embedding of s to the mean of the original vocabulary embeddings:

$$E[s] = \frac{1}{|V|} \sum_{v \in V} E[v], \tag{16}$$

where V is the original vocabulary and E is the input embedding matrix. When the input and output embeddings are not tied, we initialize the corresponding output embedding by the same rule. The replay-token embeddings remain frozen. During evaluation, we prepend s to prompts for every checkpoint trained with the replay token so that training and evaluation use the same prompt format.

Unconditional Replay Data Generation. For each task $t > 1$, the teacher generates N_R sequences:

$$\widetilde{z}_m = (\widetilde{z}_{m,1}, \ldots, \widetilde{z}_{m,L_m}) \sim p_{\bar{\Theta}_{t-1}}(\cdot \mid s), \qquad m = 1, \ldots, N_R, \tag{17}$$

where $\widetilde{z}_m$ is the mth generated sequence and L_m is its length. We use nucleus sampling with top-$p = 0.9$ and generation temperature τ_G. Generation stops at the first end-of-sequence token or after $L_{\max}$ new tokens. In our experiments, we use $N_R = 300$, $\tau_G = 1.5$, and $L_{\max} = 384$.

Each generation starts from the same one-token prompt s, with no query, answer, task index, or other information from an earlier example. Random sampling provides variation across generations. We call this replay unconditional because generation is not conditioned on an example or task identifier.

We discard empty continuations and collect the remaining samples in the temporary replay set

$$\widetilde{\mathcal{D}}_t = \{(s, \widetilde{z}_m) : m = 1, \ldots, N_R,\ L_m > 0\}. \tag{18}$$

Thus $|\widetilde{\mathcal{D}}_t| \le N_R$.

Soft replay loss. For each retained continuation, index the seed by $\ell = 0$, the continuation tokens by $\ell = 1, \ldots, L_m$, and any padding positions by $\ell > L_m$. Define the replay mask

$$a_{m,\ell} = \mathbf{1}[1 \le \ell \le L_m]. \tag{19}$$

At position ℓ, the teacher and student receive the same prefix $(s, \widetilde{z}_{m,<\ell})$. Let

$$\bar{h}_{m,\ell} = h_{\bar{\Theta}_{t-1}}(s, \widetilde{z}_{m,<\ell}), \qquad h_{m,\ell} = h_\Theta(s, \widetilde{z}_{m,<\ell}), \tag{20}$$

denote their logits for the next token. For replay temperature τ_D, define

$$q^D_{m,\ell} = \mathrm{softmax}\big(\bar{h}_{m,\ell}/\tau_D\big), \qquad p^D_{m,\ell} = \mathrm{softmax}(h_{m,\ell}/\tau_D). \tag{21}$$

The data anchor minimizes the mask-normalized forward KL over the full vocabulary:

$$\mathcal{L}^t_D = \frac{\tau_D^2}{\sum_{m,\ell} a_{m,\ell}} \sum_{m,\ell} a_{m,\ell} \mathrm{KL}\big(q^D_{m,\ell} \,\|\, p^D_{m,\ell}\big). \tag{22}$$

The factor τ_D^2 keeps the gradient scale comparable across replay temperatures.

The previous model determines both the replay sequences and the distributions that the student matches. At each generated-token position, the sampled continuation supplies a common prefix to the teacher and student, and $q^D_{m,\ell}$ is the teacher's next-token distribution. This soft target retains the

teacher's uncertainty instead of treating the sampled token as the only correct outcome. The loss compares the full next-token distributions of the teacher and student.

When a configuration uses replay at task $t > 1$, Equation 13 becomes

$$\mathcal{L}^t_{\text{fit}} = (1 - w)\mathcal{L}^t_{\text{SFT}} + w\mathcal{L}^t_D, \qquad w \in [0, 1]. \tag{23}$$

The coefficients sum to one, so w directly controls the balance between fitting the current task and matching the previous model on generated sequences. Increasing w increases the replay contribution and decreases the current task contribution by the same amount.

Training procedure. Algorithm 1 summarizes training for one task. Each epoch begins with a shuffled replay iterator. For every current minibatch, we draw one replay minibatch and create another shuffled iterator if the current one is exhausted. The current and replay losses therefore appear in the same number of minibatch updates. When the function anchor is active, we evaluate it only on the current minibatch.

Algorithm 1: Unconditional generative replay for task t. The previous model generates the replay set before the student receives its first update on task t.

Input: task data $\mathcal{D}_t$; student Θ; previous model $\bar{\Theta}_{t-1}$ when $t > 1$; seed s; attempts N_R; generation temperature τ_G; maximum length $L_{\max}$; replay temperature τ_D; replay weight w; epochs E
Output: updated student parameters

```
if t = 1 then
    train on L^1_SFT                                              // a^1_D = 0
else
    freeze Θ̄_{t-1}
    D̃_t ← ∅
    for m ← 1 to N_R do
        sample z̃_m ~ p_{Θ̄_{t-1}}(· | s) with temperature τ_G, top-p = 0.9, and limit L_max
        if z̃_m ≠ ∅ then
            add (s, z̃_m) to D̃_t
        end
    end
    for e ← 1 to E do
        reshuffle the replay iterator
        forall current minibatches B ⊂ D_t do
            draw replay minibatch B̃ ⊂ D̃_t    // recycle the iterator if needed
            L ← (1 − w)L^t_SFT(Θ, B) + wL^t_D(Θ, B̃)
            add the active function, weight, and low-rank allocation terms from Equation 14
            take one optimizer step
        end
    end
    discard D̃_t and release Θ̄_{t-1}
end
```

Relation to prior formulations. Deep Generative Replay introduced a generator and solver construction in which generated examples replace unavailable earlier data (Shin et al., 2017). LAMOL uses one autoregressive language model to solve tasks and generate pseudo-samples. It trains each example in separate question-answering and generation formats, and studies both a shared generation token and task-specific generation tokens (Sun et al., 2019). Our data anchor uses one shared token in its training and evaluation format and never supplies task identity during generation. It also matches the teacher's full next-token distributions instead of maximizing the likelihood of sampled replay tokens.

A.2.4 FUNCTION ANCHOR: PREVIOUS-STATE SELF-DISTILLATION

The function anchor uses standard forward-KL distillation to limit changes in the model's output distributions. Inspired by Learning without Forgetting, which evaluates a previous model on current task inputs when earlier inputs are unavailable (Li & Hoiem, 2017) we apply this idea by comparing distributions over the full vocabulary of LLMs at each token position.

At the start of task $t > 1$, we freeze the model obtained at task $t-1$ as $\bar{\Theta}_{t-1}$. For a current minibatch $\mathcal{B}$, let n index a sequence $z_n = (z_{n,1}, \ldots, z_{n,L_n})$, and let $\ell = 1, \ldots, L_{n-1}$ index the prefix that predicts $z_{n,\ell+1}$. We write $h_{n,\ell} = h_\Theta(z_{n,\leq\ell})$ for the student logits and $\bar{h}_{n,\ell} = h_{\bar{\Theta}_{t-1}}(z_{n,\leq\ell})$ for the teacher logits. At temperature τ_F, define

$$q^F_{n,\ell} = \mathrm{softmax}(\bar{h}_{n,\ell}/\tau_F), \qquad p^F_{n,\ell} = \mathrm{softmax}(h_{n,\ell}/\tau_F). \tag{24}$$

Here $q^F_{n,\ell}$ is the teacher distribution and $p^F_{n,\ell}$ is the student distribution. Using the sequence mask $m_{n,\ell+1}$ from Equation 11, we average the forward KL over every nonpadding target position across the full vocabulary:

$$\mathcal{L}^t_F = \frac{\tau_F^2}{\sum_{n,\ell} m_{n,\ell+1}} \sum_{n,\ell} m_{n,\ell+1} \mathrm{KL}\big(q^F_{n,\ell} \,\|\, p^F_{n,\ell}\big)\,. \tag{25}$$

At each task boundary, we replace the teacher with the model that has just completed training. The teacher therefore is the model trained on task $t-1$. Shared LoRA freezes a copy of the previous adapter. Merged LoRA and sequential OSRM use the committed dense weights with a zero residual from the new adapter. O-LoRA uses the accumulated frozen factors and holds the new B_t at zero during teacher evaluations. We run the teacher in evaluation mode, compute no gradients through it, and release it after the task.

The function and data anchors differ in the inputs on which they compare teacher and student. Equation 25 uses only sequences from the current task and generates no additional inputs. The data anchor instead samples sequences from the previous model and evaluates its loss on those generated sequences. When a configuration uses both anchors, they may share the same frozen teacher, but each anchor evaluates it on its own input set.

A.2.5 WEIGHT ANCHORS

A weight anchor limits changes to trainable coordinates that earlier tasks marked as important. Let $\vartheta = (\vartheta_1, \ldots, \vartheta_P)$ denotes the P tracked scalar coordinates. In our experiments, each coordinate is one scalar entry of a trainable LoRA factor. The implementation identifies each coordinate by its parameter name and position within the tensor. Both weight anchors use the following diagonal quadratic penalty:

$$\mathcal{R}^{t+1} = \sum_{i=1}^{P} \Omega_{t,i} \left(\vartheta_i - \vartheta^\star_{t,i}\right)^2, \tag{26}$$

where $\vartheta^\star_{t,i}$ is the reference value stored after task t and $\Omega_{t,i}$ measures the importance of coordinate i. Synaptic Intelligence (SI) estimates this importance from the optimization path during training. Online EWC estimates it from gradients of the trained model after training on the task. In the online EWC penalty, the running Fisher value $\widetilde{F}_{t,i}$ takes the place of $\Omega_{t,i}$ in Equation 26.

The general quadratic in Equation 5 matches the implemented penalties by setting $H = 2\,\mathrm{diag}(\Omega)$ for SI and $H = 2\,\mathrm{diag}(\widetilde{F})$ for online EWC. The coefficients λ_{SI} and λ_{EWC} absorb this constant factor. Section A.2.5 explains the consequences of attaching each importance value to a named LoRA coordinate.

Synaptic Intelligence. SI assigns importance according to how much each coordinate contributed to reducing the fit loss during optimization (Zenke et al., 2017). Let K_t be the number of optimizer steps on task t, and let $\vartheta_{t,k,i}$ be coordinate i immediately before step k. Thus $\vartheta_{t,0,i}$ is its value at the start of the task and $\vartheta_{t,K_t,i}$ is its value after the final step. For each optimizer step, SI accumulates the gradients from the microbatches in that step and records

$$g_{t,k,i} = \frac{\partial \mathcal{L}^t_{\mathrm{fit}}}{\partial \vartheta_{t,k,i}}, \tag{27}$$

the gradient of the fit term with respect to coordinate i. The implementation computes this gradient before adding the function and weight penalties. It then captures the unclipped gradient, clips the complete-objective gradient in Equation 14, and updates the parameters. SI therefore pairs an unclipped fit gradient with the parameter change, which can reflect every active objective term and gradient clipping.

SI accumulates the coordinatewise path contribution over task t:

$$\omega_{t,i} = -\sum_{k=0}^{K_t-1} g_{t,k,i} \left(\vartheta_{t,k+1,i} - \vartheta_{t,k,i}\right). \tag{28}$$

The summand is positive when a coordinate moves against its gradient diection, which is the local direction that reduces the loss. Thus $\omega_{t,i}$ gives a first order estimate of how much coordinate i contributed to reducing the fit loss along the optimization path.

Let

$$\Delta_{t,i} = \vartheta_{t,K_t,i} - \vartheta_{t,0,i} \tag{29}$$

denote the total displacement of coordinate i after we train the model on task t. Starting from $\Omega_{0,i} = 0$, SI updates the cumulative importance and the reference before we start training on the next task:

$$\Omega_{t,i} = \Omega_{t-1,i} + \frac{\max(0, \omega_{t,i})}{\Delta_{t,i}^2 + \xi}, \qquad \vartheta_{t,i}^\star = \vartheta_{t,K_t,i}. \tag{30}$$

Dividing by $\Delta_{t,i}^2$ gives more importance to a coordinate that reduced the loss with less movement. The constant $\xi > 0$ keeps the estimate stable when the total displacement is close to zero. Clamping the estimate at zero prevents the current task from assigning negative importance and reducing the penalty carried over from earlier tasks. SI combines the importance estimates from all completed tasks and uses the latest parameter values as the reference. It stores no task-specific states, so its retained state size does not grow with the number of tasks.

Starting with task $t + 1$, SI uses

$$\mathcal{R}_{\mathrm{SI}}^{t+1} = \sum_{i=1}^{P} \Omega_{t,i} \left(\vartheta_i - \vartheta_{t,i}^\star\right)^2. \tag{31}$$

When the data anchor is active, this term contains both the current task loss and the replay loss in Equation 13. SI therefore attributes the update to both sources. O-LoRA is the exception: its SI state records only the weighted current data loss, although the optimizer also uses the replay loss and the active regularizers. O-LoRA also resets its trainable coordinate frame at each task boundary. Section A.2.5 discusses the effect of this reset.

Online EWC. EWC uses Fisher information to measure how strongly the trained model's predictions depend on each coordinate (Kirkpatrick et al., 2017). We use the online update from Schwarz et al. (2018). It combines information from completed tasks in one running Fisher, so its retained state size does not grow with the number of tasks.

At the end of task t, let $N_t = |\mathcal{D}_t|$ be the number of training sequences, let N_{F} be the maximum number of sequences used for Fisher estimation, and let $M_t = \min(N_t, N_{\mathrm{F}})$. After training on task t, we estimate the diagonal Fisher using the first M_t sequences. For each sequence, we compute its causal language modeling loss with the model in evaluation mode and gradients enabled. Let $\ell_{t,n}$ denote the loss for sequence n. The diagonal Fisher estimate for task t is

$$F_{t,i} = \frac{1}{M_t} \sum_{n=1}^{M_t} \left(\frac{\partial \ell_{t,n}}{\partial \vartheta_i}\right)^2. \tag{32}$$

Task Fishers can differ greatly in overall magnitude. Following Schwarz et al. (2018), we normalize each task Fisher before adding it to the running estimate so that relative parameter importance, rather than raw Fisher magnitude, determines its contribution. We use the first task's mean Fisher as the common scale. Define

$$\bar{F}_t = \frac{1}{P} \sum_{i=1}^{P} F_{t,i}, \qquad \widehat{F}_{t,i} = F_{t,i} \frac{\bar{F}_1}{\bar{F}_t}, \tag{33}$$

where $\bar{F}_t$ is the mean Fisher for task t and $\bar{F}_1$ fixes the reference scale. This rescaling preserves the relative importance of coordinates within each task while keeping all task Fishers on the raw scale established by the first task. Scaling each Fisher to unit mean would instead change the effective strength of λ_{EWC} by several orders of magnitude in our setting.

Starting from $\widetilde{F}_{0,i} = 0$, online EWC uses a decay coefficient $\gamma \in [0, 1]$ and updates the running Fisher and the reference as

$$\widetilde{F}_{t,i} = \gamma \widetilde{F}_{t-1,i} + \widehat{F}_{t,i}, \qquad \vartheta^{\star}_{t,i} = \vartheta_{t,K_t,i}. \tag{34}$$

Task $t + 1$ then uses

$$\mathcal{R}^{t+1}_{\mathrm{EWC}} = \sum_{i=1}^{P} \widetilde{F}_{t,i} \left(\vartheta_i - \vartheta^{\star}_{t,i}\right)^2. \tag{35}$$

The decay coefficient γ appears only in Equation 34 because the running Fisher already contains the decay. Applying γ again in the penalty would reduce earlier contributions twice. Our experiments use $\gamma = 1$, so the running Fisher becomes a sum of the normalized task Fishers and retains all earlier contributions.

Training procedure. Algorithm 2 shows when the two anchors collect and update their state. SI records information during ordinary training and requires no additional data pass. Online EWC estimates its Fisher after training and requires one backward pass for each selected sequence. Both methods consolidate the trained parameters before a LoRA merge or an O-LoRA fold changes the current factors. This order ensures that the new importance values and reference describe the coordinates used to learn task t.

Algorithm 2: Weight-anchor update at task t. SI forms importance from the optimization path, while online EWC forms it from gradients of the trained model.

Input: task data $\mathcal{D}_t$; state $(\Omega_{t-1}, \vartheta^{\star}_{t-1})$ or $(\widetilde{F}_{t-1}, \vartheta^{\star}_{t-1})$; damping ξ; decay γ; Fisher budget N_{F}
Output: updated anchor state
snapshot $\vartheta_{t,0}$; set $\omega_t \leftarrow 0$ `// SI only`
for $k \leftarrow 0$ **to** $K_t - 1$ **do**
  accumulate $g_{t,k}$ from the fit term before adding penalties
  backpropagate the complete objective over the same accumulation window
  adjust an incomplete final window; capture $g_{t,k}$; clip the complete-objective gradient
  take one optimizer step
  $\omega_t \leftarrow \omega_t - g_{t,k} \odot (\vartheta_{t,k+1} - \vartheta_{t,k})$ `// SI only`
end
if *SI* **then**
  $\Omega_t \leftarrow \Omega_{t-1} + \max(0, \omega_t) \oslash (\Delta_t^{\odot 2} + \xi)$
else if *online EWC* **then**
  estimate F_t from $\min(N_t, N_{\mathrm{F}})$ individual sequences
  rescale F_t to $\widehat{F}_t$
  $\widetilde{F}_t \leftarrow \gamma \widetilde{F}_{t-1} + \widehat{F}_t$
$\vartheta^{\star}_t \leftarrow \vartheta_{t,K_t}$ `// before a merge or fold`

Coordinate dependence of weight anchors. After each task, SI and online EWC save the current LoRA parameters as a reference and estimate how important each parameter is for retaining what the model has learned. Shared LoRA keeps the same A and B matrices across tasks, so Equations 31 and 35 compare each entry with its earlier value. Merged LoRA instead adds the completed update $B_t A_t$ to the dense weight matrix and creates new A and B matrices with the same names and dimensions. Sequential OSRM follows the same procedure but uses its own rule to initialize A. O-LoRA moves the completed update into its frozen accumulated matrices and then reinitializes the trainable A and B matrices without changing their names. After these resets, our implementation matches each stored reference and importance weight to the entry at the same row and column in the new matrices. An old importance weight therefore applies to a newly initialized parameter, even though changing that parameter may now affect the dense weight differently. This procedure defines a numerical penalty, but it does not preserve the original meaning of the importance weights across a reset.

To preserve the previous penalty after changing the parameter representation, we would need to transform both its reference values and its importance matrix. Suppose an invertible map $\vartheta_{\text{old}} = f_t(\vartheta_{\text{new}})$ relates the two representations. The new reference $\vartheta^\star_{\text{new}}$ must satisfy $f_t(\vartheta^\star_{\text{new}}) = \vartheta^\star_{\text{old}}$. Let J_t denote the Jacobian of f_t at $\vartheta^\star_{\text{new}}$. For small changes around the two reference points, preserving the quadratic penalty requires

$$\delta\vartheta_{\text{old}} = J_t \delta\vartheta_{\text{new}}, \qquad H_{\text{new}} = J_t^\top H_{\text{old}} J_t, \tag{36}$$

where H_{old} and H_{new} contain the importance weights in the two representations. This transformation is exact when f_t is linear and describes the local behavior near the reference when f_t is nonlinear. Even if H_{old} is diagonal, H_{new} generally contains off-diagonal terms that connect changes in different parameters. SI and online EWC store only one importance weight per parameter, so they cannot fully represent these interactions. When a low-rank allocation rule resets the LoRA workspace, it initializes new A and B matrices. Our implementation keeps the SI or online EWC reference and importance tensors and matches each stored value to the parameter at the same row and column in the new matrices. Since we don't have a transformation defined above, we do not adjust these stored values for the change in A and B. Thus, the penalties applied to the new LoRA matrices may not have the same effect when they are applied to a dense model.

A.2.6 Low-Rank Allocation Rules

Let $\mathcal{J}$ denote the set of linear transformations to which we apply LoRA. For target $j \in \mathcal{J}$, let $W_{0,j} \in \mathbb{R}^{d_{\text{out},j} \times d_{\text{in},j}}$ be the pretrained weight matrix, and let $A_{t,j} \in \mathbb{R}^{r \times d_{\text{in},j}}$ and $B_{t,j} \in \mathbb{R}^{d_{\text{out},j} \times r}$ denote the LoRA matrices for module j during task t, with LoRA rank r. The LoRA scale is $\rho = \alpha_{\text{LoRA}}/r$. We write $(A^{(0)}_{t,j}, B^{(0)}_{t,j})$ for the LoRA matrices at the start of task t and $(A^\star_{t,j}, B^\star_{t,j})$ for their values after training on that task. The matrix $W^{\text{eff}}_{t,j}$ denotes the effective dense weight of module j after task t, obtained by combining the base weight with all LoRA updates active in evaluation mode. The low-rank allocation rules differ in whether they reuse, merge, or retain the task factors.

Shared LoRA. Shared LoRA uses one LoRA while we train it over all the tasks (Hu et al., 2022). Let Opt_t denote all AdamW updates performed on task t, including the task-specific learning-rate schedule and every active term in Equation 14. During task t,

$$W^{\text{eff}}_{t,j} = W_{0,j} + \rho B_{t,j} A_{t,j}, \qquad (A^\star_{t,j}, B^\star_{t,j}) = \text{Opt}_t\left(A^{(0)}_{t,j}, B^{(0)}_{t,j}\right). \tag{37}$$

For the first task, the implementation initializes $A^{(0)}_{1,j}$ with Kaiming initialization and sets $B^{(0)}_{1,j} = 0$. For each later task, it sets $(A^{(0)}_{t,j}, B^{(0)}_{t,j}) = (A^\star_{t-1,j}, B^\star_{t-1,j})$. It creates a new optimizer and learning-rate schedule at every task boundary but keeps the learned factors. Thus, the cumulative adaptation of each target weight matrix has rank at most r, regardless of the number of tasks. Plain sequential supervised fine-tuning uses this low-rank allocation rule with every anchor indicator in Equation 14 set to zero.

merged LoRA. Merged LoRA uses the merge and restart pattern associated with ReLoRA (Lialin et al., 2024). ReLoRA schedules several restarts during pretraining and partially resets the optimizer state. This low-rank allocation rule instead performs one merge at each task boundary and creates a new optimizer for the next task.

Let $W_{t-1,j}$ denote the dense weight matrix obtained after merging the LoRA updates learned from tasks 1 through $t-1$. Task t uses a rank-r LoRA:

$$W^{\text{eff}}_{t,j} = W_{t-1,j} + \rho B_{t,j} A_{t,j}. \tag{38}$$

After training, the method folds the final task factors into the dense matrix:

$$W_{t,j} = W_{t-1,j} + \rho B^\star_{t,j} A^\star_{t,j}. \tag{39}$$

The implementation then removes the old adapter and attaches a new one with $A^{(0)}_{t+1,j}$ initialized by the Kaiming rule and $B^{(0)}_{t+1,j} = 0$. Because the new LoRA update is initialized to zero, $B_{t+1,j} A_{t+1,j} = 0$, task $t+1$ starts from the function represented by the merged weight $W_{t,j}$. Each task contributes a matrix of rank at most r, so the sum of the committed residuals can reach rank tr.

O-LoRA. O-LoRA assigns a new rank-r LoRA to each task and keeps all earlier pairs fixed (Wang et al., 2023a). Our implementation follows the released two-pair structure: one concatenated pair stores completed tasks, while one fixed-rank pair learns the current task. The original method applies LoRA to the query and value projections. For a controlled comparison, we apply every low-rank allocation rule to the common target set in Table 3.

At the start of task t, define the accumulated factors

$$A_{<t,j} = \begin{bmatrix} A^\star_{1,j} \\ \vdots \\ A^\star_{t-1,j} \end{bmatrix}, \qquad B_{<t,j} = \begin{bmatrix} B^\star_{1,j} & \cdots & B^\star_{t-1,j} \end{bmatrix}. \tag{40}$$

The represented target matrix is

$$W^{\text{eff}}_{t,j} = W_{0,j} + \rho B_{<t,j} A_{<t,j} + \rho B_{t,j} A_{t,j}. \tag{41}$$

Only $(A_{t,j}, B_{t,j})$ receives gradients. The accumulated LoRAs remain frozen. At the start of the task, we initialize $A_{t,j}$ with the Kaiming rule and sets $B_{t,j} = 0$.

O-LoRA encourages the current input-side factors to differ from the accumulated input-side factors. The implementation uses

$$\mathcal{R}^t_\perp = \sum_{j \in \mathcal{J}} \left\| A_{<t,j} A^\top_{t,j} \right\|_{1,1}, \qquad \|C\|_{1,1} = \sum_{a,b} |C_{a,b}|. \tag{42}$$

This penalty is zero on the first task because no earlier LoRAs exist. The released implementation uses the entrywise ℓ_1 norm in Equation 42, whereas the paper presents a squared Frobenius penalty. We follow the released implementation so the experiment matches its executable training rule.

The original code also provides the optional factor norm

$$\mathcal{R}^t_2 = \sum_{j \in \mathcal{J}} \left(\|A_{t,j}\|_F + \|B_{t,j}\|_F \right). \tag{43}$$

The regular configuration sets $\lambda_2 = 0$, so this term does not affect training. Our implementation adds $\mathcal{R}^t_\perp$ and $\mathcal{R}^t_2$ without dividing them by the gradient-accumulation factor, following the released code.

After task t, the method appends $(A^\star_{t,j}, B^\star_{t,j})$ to the accumulated LoRA set and start a new LoRA for training on the next task. The model after task t therefore represents

$$W_{t,j} = W_{0,j} + \rho \sum_{s=1}^{t} B^\star_{s,j} A^\star_{s,j}. \tag{44}$$

The evaluation code reconstructs the same matrix by adding the saved rank-r LoRA for tasks 1 through t to a fresh base model. It does not select adapters using a task identifier.

Sequential OSRM adaptation. OSRM chooses each LoRA input subspace from features of the other tasks before fine-tuning and then merges the independently trained task models (Zhang & Zhou, 2025). This procedure assumes access to training data from all tasks. Our sequential adaptation cannot use future tasks, so it uses features only from completed tasks to initialize the next LoRA workspace.

For a completed task s, let $N_s = |\mathcal{D}_s|$, let N_H be the feature-sample budget, and let $M^H_s = \min(N_s, N_H)$. The implementation takes the first M^H_s datapoints in the task data order and divides them into C_s forward batches. For each LoRA-adapted linear module j, let $h_{j,\ell}(z) \in \mathbb{R}^{d_{\text{in},j}}$ denote the input hidden-state vector at sequence position ℓ, measured immediately before the module applies the LoRA down-projection matrix A_j while processing the training sequence z. Let $\mathcal{B}_{s,b}$ be feature batch b and let $L_{s,b}$ be its padded sequence length. The stored feature is

$$\bar{h}_{s,j} = \frac{1}{C_s} \sum_{b=1}^{C_s} \frac{1}{|\mathcal{B}_{s,b}| L_{s,b}} \sum_{z \in \mathcal{B}_{s,b}} \sum_{\ell=1}^{L_{s,b}} h_{j,\ell}(z). \tag{45}$$

We first average over positions and sequences within each batch and then gives every batch mean equal weight. Before task t, the method stacks the feature vectors from completed tasks:

$$H_{<t,j} = \begin{bmatrix} \bar{h}_{1,j}^\top \\ \vdots \\ \bar{h}_{t-1,j}^\top \end{bmatrix} \in \mathbb{R}^{(t-1)\times d_{\text{in},j}}. \tag{46}$$

Let $H_{<t,j} = U\Sigma V^\top$ be its full singular value decomposition, and let $V_{\min} \in \mathbb{R}^{d_{\text{in},j}\times r}$ contain the r right singular vectors associated with the smallest singular values. Let $\widehat{A}_{t,j}$ denote the ordinary Kaiming initialization drawn when the implementation attaches the fresh adapter. Define

$$\kappa_{t,j} = \frac{1}{r}\sum_{a=1}^{r} \left\| \widehat{A}_{t,j}[a,:] \right\|_2, \qquad A_{t,j}^{(0)} = \kappa_{t,j} V_{\min}^\top, \qquad B_{t,j}^{(0)} = 0. \tag{47}$$

The smallest right singular vectors identify input directions with the least energy in the stored past-task means. When the null space of $H_{<t,j}$ has dimension at least r, the selected rows lie in that null space and satisfy $A_{t,j}^{(0)} \bar{h}_{s,j} = 0$ for every stored task $s < t$. The factor $\kappa_{t,j}$ restores the mean row norm of the ordinary Kaiming initialization because unscaled singular vectors have unit norm.

The first task uses ordinary LoRA initialization because it has no past features. After each later task, the implementation first collects $\bar{h}_{t,j}$ from the trained model, then merges the learned LoRA matrices with Equation 39, and finally initializes the next fresh LoRA from all stored feature means. Relative to the original post-training merge setting, this version makes three changes: it uses only past features, extracts them from the just-trained model rather than a common pretrained model, and rescales the selected directions before optimizing both factors. Thus, we refer to it as a sequential OSRM adaptation.

A.2.7 Training Procedure

Algorithm 3 records the training procedure. The order of combining different algorithms matters because teachers must describe the model before task t, while the weight anchors and OSRM features must describe the model after it learns task t but before a low-rank allocation rule changes its coordinates.

Algorithm 3: Order of the operations at the boundary of task t.

Input: task data $\mathcal{D}_t$; learner state after task $t-1$; active anchors; low-rank allocation rule
Output: learner state after task t
load and format $\mathcal{D}_t$
if $t > 1$ **then**
  construct each active previous-state teacher
  generate the fixed-budget replay set if the data anchor is active
end
snapshot the trainable starting coordinates if SI is active
optimize Equation 14 on task t
consolidate SI if it is active; estimate the online-EWC Fisher if it is active
collect $\{\bar{h}_{t,j} : j \in \mathcal{J}\}$ from the trained model if sequential OSRM is active
if *shared LoRA* **then**
  retain $(A_{t,j}^\star, B_{t,j}^\star)$ for task $t+1$
else if *merged LoRA* **then**
  commit Equation 39; attach a fresh rank-r workspace
else if *O-LoRA* **then**
  save the current rank-r factors; append them to the frozen accumulated pair; reset the workspace
else
  commit Equation 39; attach a fresh workspace; apply Equation 47 `// sequential OSRM`
end

To compare persistent method state, let

$$P_r = \sum_{j \in \mathcal{J}} r \left(d_{\mathrm{in},j} + d_{\mathrm{out},j} \right) \tag{48}$$

be the number of scalars in one rank-r adapter. This count excludes evaluation checkpoints and diagnostic logs because they do not form part of the state used to learn later tasks. Shared LoRA stores one P_r LoRA adapter. Merged LoRA stores one dense model and one P_r LoRA adapter, so its additional adapter state does not grow with the number of completed tasks. After t completed tasks, O-LoRA stores tP_r accumulated scalars and one P_r LoRA adapter while training the next task. Sequential OSRM stores the merged LoRA state together with one $d_{\mathrm{in},j}$-dimensional feature vector for every completed task and target, so its retained feature state has size $O(t \sum_{j \in \mathcal{J}} d_{\mathrm{in},j})$.

A.2.8 Optimization and Method Settings

Table 3 records the settings used by the registered canonical runs. For hyperparameters selected by task-level successive halving, the table gives both the candidate set and the registered value.

Component	Setting
Backbone	Qwen3-4B-Base
Current-task optimization	10 epochs per task; minibatch size 8; gradient accumulation 1; maximum sequence length 384
Optimizer	AdamW; learning rate 5×10^{-4}; weight decay 0.01; global gradient-norm clipping at 1
Per-task schedule	Linear warmup during the first 5% of optimizer steps, followed by a constant learning rate; optimizer and schedule restart for every task
LoRA	$r = 32$; $\alpha_{\mathrm{LoRA}} = 64$; $\rho = 2$; dropout 0.05; target modules {`q_proj`, `k_proj`, `v_proj`, `o_proj`, `gate_proj`, `up_proj`, `down_proj`}
Data anchor	$N_R = 300$; $\tau_D = 2$; frozen replay seed token; top-$p = 0.9$; generation batch size 32; maximum 384 new tokens; generation-temperature candidates $\{1.0, 1.5\}$ and registered value 1.5; w candidates $\{0.5, 0.75\}$ and registered values 0.75 (SYMBOL-QA), 0.5 (LLM-QA), and 0.75 (REAL-QA)
Function anchor	Forward KL over the complete vocabulary; $\tau_F = 5$; λ_F candidates $\{1, 3\}$ and registered value 1; previous-state teacher refreshed at every task
SI	$\lambda_{\mathrm{SI}} = 1$; $\xi = 0.1$; negative task contributions clamped to zero
Online EWC	$\lambda_{\mathrm{EWC}} = 1000$; $\gamma = 1$; at most $N_{\mathrm{F}} = 1000$ sequences per task; normalization to the first task's mean Fisher enabled
O-LoRA	$\lambda_{\perp} = 0.5$; $\lambda_2 = 0$; rank 32 added for each task
Sequential OSRM	Up to $N_H = 64$ training sequences are used to compute one mean input-feature vector per adapted layer and task. Padded positions are included. Each new A is initialized from the smallest right singular vectors and rescaled to the mean row norm of its Kaiming initialization

Table 3: Canonical implementation and optimization settings.

The canonical suite uses unconditional KL replay with a frozen replay-token embedding and forward-KL function distillation over the complete vocabulary. It does not use query-conditioned replay. It also leaves the optional replay cross-entropy, trainable replay-token embedding, reverse KL, Jensen-Shannon divergence, vocabulary truncation, O-LoRA gradient projection, and O-LoRA factor-norm paths disabled. We therefore do not treat these options as separate methods in Table 2.

A.3 DATA CONSTRUCTION

This section documents how we build three datasets for Section 4. All three share the same interface: a dataset consists of 100 tasks, each holding a list of texts `Question: <q>\nAnswer: <a>` whose test split is a copy of its train split. At training and evaluation time the answer is rendered as `\boxed{a}`, so the delimiter is learned as a format norm (Appendix A.2.8).

A.3.1 DATA UNIQUENESS

A checkpoint trained through task i is queried with $z_{j,n}$ alone for all $j \leq i$. If two tasks contained the same question with different answers, the target would be undefined and forgetting would be confounded with ambiguity. For the synthetically generated datasets, each generator therefore maintains a registry of emitted question keys and rejects any candidate whose key has already appeared. SYMBOL-QA constrains keys only, since the key is what the model is queried on. LLM-QA constrains both the invented entity and the full question string.

A.3.2 SYMBOL-QA

SYMBOL-QA contains 10,000 random key–value associations divided evenly across 100 tasks. Each task contains 100 items. For each item, the generator samples a six-character key and a four-character value from the same 62-symbol alphabet of uppercase letters, lowercase letters, and digits. It redraws any key that has already appeared, ensuring that every key maps to a single value across the full dataset. All tasks use the same alphabet, lengths, and question format, so an item's surface form does not reveal its task. Because the generator selects each value independently of its key, there are no general patterns associating the keys to their answers. The model must therefore remember each key– value association. Training and evaluation use the same associations, so this dataset measures retention rather than generalization to unseen pairs.

A.3.3 LLM-QA

Generation. We use the instruction-tuned Qwen3-4B-Instruct-2507 model (Yang et al., 2025) to generate facts about fictional entities. The dataset defines 100 topics, such as "fictional lighthouses and their keepers," and assigns one topic to each task. For every prompt, we ask the generator to produce ten candidate JSON records with the fields `entity`, `question`, and `answer`. The prompt instructs the model to invent entities and facts that do not exist and to avoid real people, places, organizations, works, and events.

Validation. For each candidate, the validator extracts three non-empty fields and checks that the question contains the entity name after normalizing case and whitespace. This requirement makes the fictional entity the main information that distinguishes one question from another. The validator also limits answers to eight words and rejects questions or answers that contain line breaks. Across the full dataset, the validator rejects any candidate whose normalized entity or question matches one already accepted. Generation continues until every task contains 100 valid and globally distinct questions.

Task order. The original topic list places related subjects near one another. After generation, we apply a fixed random permutation with seed 0 to break this ordering. The permutation changes only the task indices, and it leaves every entity, question, answer, and topic unchanged.

Novelty. The generation prompt asks for fictional entities and facts, but the pipeline does not verify their nonexistence against an external source. We therefore do not claim that every LLM-QA item was empirically unknown to the base model before training.

A.3.4 REAL-QA

Sources. We draw equally from ten public English question-answering datasets, five short-answer and five multiple-choice (Table 4). Short-answer sources are TriviaQA (Joshi et al., 2017), the open-domain formulation (Lee et al., 2019) of Natural Questions, PopQA (Mallen et al., 2023), SQuAD (Rajpurkar et al., 2016), and WebQuestions (Berant et al., 2013). Multiple-choice sources

are OpenBookQA (Mihaylov et al., 2018), SciQ (Welbl et al., 2017), both partitions of ARC (Clark et al., 2018), and MedMCQA (Pal et al., 2022).

Normalization to free-form recall. Each source is mapped to a common record with a question, a primary answer, and an alias list. Multiple-choice items are converted by discarding the options and keeping the text of the correct choice as the target, so the model must produce the answer rather than select a letter. For datasets that consist of multiple correct answer aliases, we accept all of them as the correct answer.

Per-model contamination filter. Facts the base model already knows cannot be acquired, and would enter the accuracy matrix as spurious retention. For each candidate, we prompt the base model with `Question: <q>\nAnswer:` and draw five samples (temperature 0.7, top-$p = 0.95$, at most 64 new tokens, stopping at a newline or a new question). A candidate is discarded if any of the five completions contains any accepted answer as a word-boundary substring, while the remainder are retained. Word boundaries matter because short aliases such as country codes otherwise match inside unrelated words. Retained items are therefore items the base model failed to produce under five attempts. We do not claim the complete absence of knowledge from the model's pretraining phase, but we treat this as an indication of the model's less familiar facts for it to learn in our training process.

Assembly. From each source we sample 500 filtered items with a fixed seed, giving 5000 items. Sampling equally from ten sources rather than proportionally prevents the largest corpus from dominating the dataset. We then pool all 5000 items, shuffle globally under a fixed seed, and cut the shuffled sequence into 100 consecutive tasks of 50. Each task is thus a uniform mixture of all ten sources. This procedure helps us to remove possible interference from dataset domains during training. For instance, if we don't do the shuffling, the model may learn from the same original dataset in a task and thus may have a better retention rate that is hard to explain.

Source	Type	Original form	Retained
TriviaQA	short answer	trivia questions with alias sets	500
NQ-Open	short answer	search queries, short answer list	500
PopQA	short answer	entity-centric relational queries	500
SQuAD	short answer	span answers over passages	500
WebQuestions	short answer	Freebase question–answer pairs	500
OpenBookQA	multiple choice	elementary science, four options	500
SciQ	multiple choice	crowdsourced science questions	500
ARC-Easy	multiple choice	grade-school science, four options	500
ARC-Challenge	multiple choice	retrieval-resistant science questions	500
MedMCQA	multiple choice	medical entrance-exam questions	500

Table 4: REAL-QA sources.

A.3.5 WHAT THE THREE TIERS ISOLATE

The three datasets test memorization with increasing amounts of familiar language and knowledge. All three use the same question and answer format, but they differ in how much the model can use what it learned before continual training. In SYMBOL-QA, the key and value have no meaningful relationship, so the model must remember each individual association. LLM-QA uses natural questions about invented entities. Familiar language and entity types may help the model encode these examples, but real-world knowledge does not determine their answers. REAL-QA uses questions and answers about real entities and filters out questions that the base model answers correctly before training. Comparing the three datasets shows whether a method behaves similarly for arbitrary associations, fictional facts expressed in natural language, and real knowledge.

Algorithm 4: TSH over increasing task horizons. For each seed, a promoted configuration continues from its previous checkpoint instead of training the completed tasks again.

Input: candidate set $\mathcal{A}_1$, task horizons $r_1 < \cdots < r_K$, survivor counts $n_2, \ldots, n_K$, seeds $\mathcal{S}$, fixed task order π
Output: candidate configurations ranked at horizon r_K
for $k \leftarrow 1$ **to** K **do**
  forall $a \in \mathcal{A}_k$ *and* $s \in \mathcal{S}$ **do**
    if $k = 1$ **then**
      train configuration a from the pretrained model on tasks $1, \ldots, r_1$ using order π and seed s
    else
      restore the checkpoint and training state saved at horizon r_{k-1}, then train tasks $r_{k-1}{+}1, \ldots, r_k$
    end
    evaluate the resulting model on tasks $1, \ldots, r_k$ to obtain $M^{a,s}_{r_k,j}$ for $j \le r_k$
  end
  score each $a \in \mathcal{A}_k$ by $F_{r_k}(a)$ from Equation 9
  rank $\mathcal{A}_k$ in descending order of $F_{r_k}(a)$
  if $k < K$ **then**
    $\mathcal{A}_{k+1} \leftarrow$ the top n_{k+1} configurations in $\mathcal{A}_k$
  end
end
return *the configurations in* $\mathcal{A}_K$ *in ranked order*

Dataset	Tasks × items	Novelty control	Task organization	Semantic information
SYMBOL-QA	100 × 100	Randomly generated symbol associations	One shared generation rule across all tasks	No structure in the target mapping
LLM-QA	100 × 100	Fictional generation prompt without a base model filter	One topic per task with a fixed random topic order	Natural language and familiar entity types
REAL-QA	100 × 50	Five sample filter for each base model	Ten data sources mixed across tasks by a global shuffle	Natural language and real facts

Table 5: Comparison of the three memorization datasets. In every dataset, training and evaluation use the same questions, each question has one target answer across all tasks, and evaluation provides no task identifier.

A.4 SEARCH PROCEDURE DETAILS

This section describes the task-level successive halving search introduced in Section 4.3. The search ranks 90 configurations that combine shared LoRA or merged LoRA with optional weight, function, and data anchors. O-LoRA and sequential OSRM do not enter this search pool because their stored state grows with the number of tasks. We evaluate them separately as task-growing low-rank allocation rules.

A.4.1 COST ANALYSIS

Let r_k denote the cumulative number of tasks trained at search run k, and let $n_k = |\mathcal{A}_k|$ denote the number of configurations evaluated at that run. Let $n_k = |\mathcal{A}_k|$ denote the number of configurations that reach horizon r_k. We count training one configuration on one task as one unit of training cost. At run k, each of the n_k configurations trains only the $r_k - r_{k-1}$ tasks added since the previous run. The total training cost per seed and dataset is therefore

$$C = \sum_{k=1}^{K} n_k (r_k - r_{k-1}), \qquad r_0 = 0. \tag{49}$$

For the schedule in Section 4.3,

$$C = 90(10) + 45(10) + 23(30) + 10(50) = 2540. \tag{50}$$

Training all 90 configurations for 100 tasks would instead require 9000 such units per seed. The search therefore uses 28.2% of the exhaustive training cost, which corresponds to a 71.8% reduction.

A.4.2 Selection and continuation

Every configuration uses seeds 41, 42, and 43 at each horizon. We average the three scores before ranking, so one seed does not determine whether a configuration advances. After 10 tasks, the search retains the top $\lceil 90/2 \rceil = 45$ configurations. After 20 tasks, it retains the top $\lceil 45/2 \rceil = 23$. After 50 tasks, it retains the top 10 for the final 100 task horizon. The schedule starts at 10 tasks because pilot runs at five tasks provided little separation among the configurations.

When a configuration advances, the trainer restores its model parameters, the state required by its continual learning methods, and the random number generator states. It reconstructs the temporary teachers for the function and data anchors from the restored previous model at the start of the next task. With the same seed, task order, and training settings, this continuation reproduces the trajectory of an uninterrupted run rather than approximating it with a smaller model or less data.

A.4.3 Implementation

For caching, we group the three seed runs for one configuration at one task budget into a search cell. Each cell records the model, dataset, task budget, method choices, method hyperparameters, fixed training settings, seed list, and task order. Before reusing a completed cell, the implementation checks that all recorded settings match the current search. This allows searches with different selection schedules to reuse previously computed results when the underlying experiments are identical. If any setting differs, the implementation aborts instead of reusing the cell. During the search, every configuration uses the task permutation generated with seed 1234. Because the cache record includes this task order, search results cannot be confused with results from the default order. After selection, we retrain the chosen configurations from the pretrained model on the default task order.

After evaluating a search cell, the implementation retains the checkpoint and continuation state saved after its last task and removes the earlier task checkpoints. These files contain everything needed to continue a promoted configuration at the next run.

A.4.4 Search outputs

For each dataset, the search writes a leaderboard and a trace containing every configuration's score, rank, and selection status at each horizon it reaches. We fix the selected configuration and its searched hyperparameters before running the final experiments. The final experiments retrain the selected configurations without further tuning and present the tasks in their numbered order for that dataset.

A.4.5 Transfer to the report task order.

The preceding analysis keeps the task order fixed. We next compare the search ranking at each stage with the 100-task ranking from the final experiments. The search uses the development task order, whereas the final experiments use a separate report task order. This comparison therefore tests whether the search ranking remains informative when both the number of tasks and their order change.

Search candidates vary in both method composition and hyperparameters, whereas the final experiments report one setting for each composition. To compare them, we group search candidates by their corresponding reported composition and represent each group by its highest-scoring hyperparameter variant at that search stage. This follows the search procedure, which promotes the best-performing variant. We include only compositions evaluated in the final experiments. Let Δ denote the difference between a composition's search rank and its final rank. Table 6 reports the agreement between the two rankings.

After 10 tasks, all 17 comparable compositions remain in the search. The Spearman correlations with the final ranking are 0.93, 0.95, and 0.90 on SYMBOL-QA, LLM-QA, and REAL-QA, respec-

tively. The mean absolute rank displacement is 1.5, 1.3, and 1.4 positions. Thus, the first search stage closely approximates the final ranking even though the final experiments use a different task order and continue training through 100 tasks.

We do not interpret the lower correlations at some later stages as evidence that the ranking becomes less reliable. Each search stage removes configurations, so the comparison set becomes smaller and increasingly consists of strong configurations with similar scores. A one-position change among these configurations can substantially change the correlation. For example, the 50-task comparison on SYMBOL-QA contains only five compositions. We therefore use the 10-task results, which cover all 17 comparable compositions, as the primary check and report the later stages for completeness.

Dataset	Tasks	Compositions	Mean $\lvert\Delta\rvert$	Max $\lvert\Delta\rvert$	ρ	Top-3 overlap
SYMBOL-QA	10	17	1.5	5	0.93	2/3
	20	9	1.3	3	0.78	2/3
	50	5	1.2	2	0.50	2/3
	100	3	0.7	1	0.50	3/3
LLM-QA	10	17	1.3	3	0.95	2/3
	20	6	0.3	1	0.94	2/3
	50	4	0.5	1	0.80	2/3
	100	3	0.7	1	0.50	3/3
REAL-QA	10	17	1.4	5	0.90	3/3
	20	8	0.0	0	1.00	3/3
	50	4	0.0	0	1.00	3/3
	100	3	0.0	0	1.00	3/3

Table 6: Agreement between the search ranking on the development task order and the final ranking on the report task order. We group search candidates by method composition and represent each composition by its highest-scoring hyperparameter variant at each search stage. The displacement $|\Delta|$ measures the absolute difference between the search and final ranks. Top-3 overlap reports how many of the three highest-ranked final compositions also appear in the top three at that search stage. The comparison set becomes smaller as the search removes candidates, so the 10-task rows provide the most complete comparison.

A.5 ADDITIONAL RESULTS

This section records everything the main text compresses: every configuration on every dataset, the complete factorial including terms that are not significant, the task-growing low-rank allocation experiments, the compositions that fail outright, the seed-variance controls, and the held-out general capability evaluation.

A.5.1 CROSS-DATASET RANKING

Table 7 ranks the 16 compositions in the main 2^4 factorial by mean final retention within each dataset. For composition m, let $r_{m,d}$ be its rank on dataset d. We define $k_m = \max_d r_{m,d}$, which is the smallest k for which the composition is among the top k on every dataset. Lower values indicate more consistent performance across datasets. We use mean rank to order compositions with the same k_m. Our best method combines all three anchors with merged LoRA. It is the only composition that ranks among the top 3 methods in all datasets, so it uniquely attains $k_m = 3$. Its average final retention is 34.9% across the three datasets.

Composition	SYMBOL-QA	LLM-QA	REAL-QA	k_m	Mean rank
All anchors + merged LoRA	**2**	**1**	**3**	**3**	**2.00**
SI + Replay + merged LoRA	4	2	1	4	2.33
SD + Replay + merged LoRA	1	3	4	4	2.67
Replay + merged LoRA	3	4	2	4	3.00
SI + SD + Replay	5	5	8	8	6.00
SD + merged LoRA	6	8	6	8	6.67
SI + Replay	8	7	7	8	7.33
SI + SD + merged LoRA	10	6	5	10	7.00
SI + SD	9	10	9	10	9.33
SD + Replay	7	9	11	11	9.00
Replay	11	11	10	11	10.67
SD	12	13	12	13	12.33
SI + merged LoRA	14	14	13	14	13.67
merged LoRA	13	12	15	15	13.33
SI	15	15	14	15	14.67
Naive fine-tuning	16	16	16	16	16.00

Table 7: Cross-dataset ranks of the 16 compositions in the main factorial evaluation. Rows are ordered by the worst rank k_m, with mean rank used to break ties.

A.5.2 COMPLETE PER-CONFIGURATION RESULTS

configuration	Final	Diag	Forget	$W(10)$	$W(50)$
`sd_replay_merge`	23.2 ± 4.2	99.5 ± 0.3	77.2 ± 4.1	95.4 ± 1.7	45.9 ± 8.1
`si_sd_replay_merge`	18.5 ± 2.8	99.3 ± 0.1	81.8 ± 2.8	93.9 ± 3.7	37.0 ± 5.5
`si_sd_replay_osrm`	17.7 ± 2.6	99.3 ± 0.5	82.8 ± 2.5	94.8 ± 0.3	35.4 ± 5.1
`replay_merge`	16.6 ± 5.8	99.9 ± 0.0	84.2 ± 5.9	89.0 ± 5.4	33.1 ± 11.6
`si_replay_merge`	15.7 ± 3.3	99.8 ± 0.2	85.0 ± 3.2	86.5 ± 8.4	31.3 ± 6.5
`si_sd_replay`	15.4 ± 0.3	99.3 ± 0.8	84.9 ± 0.8	89.7 ± 0.5	30.7 ± 0.6
`si_sd_replay_olora`	15.3 ± 4.5	98.8 ± 0.3	84.5 ± 4.7	79.2 ± 5.6	29.7 ± 8.3
`sd_merge`	11.8 ± 1.3	99.8 ± 0.1	88.9 ± 1.3	83.4 ± 4.5	23.5 ± 2.7
`sd_replay`	9.0 ± 0.5	99.6 ± 0.3	91.5 ± 0.2	78.3 ± 2.8	18.0 ± 1.0
`si_replay`	7.6 ± 0.8	99.9 ± 0.0	93.2 ± 0.8	68.5 ± 5.6	15.3 ± 1.5
`si_sd`	6.9 ± 0.6	99.3 ± 0.1	93.4 ± 0.5	60.3 ± 4.9	13.7 ± 1.2
`si_sd_merge`	6.8 ± 5.9	95.0 ± 8.0	89.3 ± 1.9	53.5 ± 46.4	13.5 ± 11.7
`replay`	4.2 ± 0.9	99.8 ± 0.1	96.6 ± 1.0	41.9 ± 8.3	8.4 ± 1.8
`sd`	3.3 ± 0.0	99.8 ± 0.1	97.5 ± 0.1	33.4 ± 0.4	6.7 ± 0.1
`olora`	1.8 ± 0.1	99.9 ± 0.1	99.1 ± 0.2	18.2 ± 1.0	3.6 ± 0.2
`merge`	1.7 ± 0.1	98.5 ± 0.6	97.8 ± 0.7	17.1 ± 0.9	3.4 ± 0.2
`osrm`	1.6 ± 0.1	98.7 ± 0.3	98.1 ± 0.3	15.5 ± 1.2	3.1 ± 0.2
`si_merge`	1.5 ± 0.2	97.7 ± 0.3	97.2 ± 0.5	14.6 ± 1.6	2.9 ± 0.3
`si`	1.3 ± 0.1	96.4 ± 0.2	96.1 ± 0.3	12.6 ± 1.0	2.5 ± 0.2
`online_ewc`	1.0 ± 0.6	86.0 ± 15.1	85.8 ± 14.6	10.3 ± 6.3	2.1 ± 1.3
`vanilla`	1.0 ± 0.1	97.4 ± 0.4	97.4 ± 0.3	10.3 ± 0.5	2.1 ± 0.1

Table 8: All configurations on SYMBOL-QA, mean $\pm$ standard deviation over three seeds. $W(k)$ is the final-checkpoint accuracy averaged over the k most recently trained tasks.

configuration	Final	Diag	Forget	$W(10)$	$W(50)$
si_replay_olora	42.2 ± 5.2	99.8 ± 0.0	58.2 ± 5.2	93.9 ± 1.9	60.2 ± 5.7
si_sd_replay_merge	41.8 ± 1.4	99.3 ± 0.1	58.3 ± 1.5	96.3 ± 0.5	65.5 ± 0.8
si_replay_merge	41.2 ± 9.5	99.6 ± 0.2	59.1 ± 9.4	98.2 ± 0.7	64.3 ± 12.1
sd_replay_merge	33.0 ± 1.8	99.4 ± 0.2	67.1 ± 1.8	94.6 ± 2.7	54.9 ± 1.9
replay_merge	32.4 ± 8.9	99.7 ± 0.1	68.0 ± 9.0	97.5 ± 2.3	54.8 ± 13.7
si_replay_osrm	29.7 ± 3.0	99.5 ± 0.1	70.6 ± 3.0	98.4 ± 0.5	51.2 ± 5.5
si_sd_replay	18.6 ± 0.8	99.5 ± 0.1	81.7 ± 0.8	86.4 ± 1.4	33.0 ± 1.7
si_sd_merge	18.3 ± 0.9	99.4 ± 0.1	81.9 ± 0.9	75.7 ± 2.1	33.4 ± 1.8
si_replay	15.5 ± 2.7	99.6 ± 0.2	84.9 ± 2.6	80.4 ± 6.1	27.6 ± 4.8
sd_merge	12.8 ± 0.8	99.6 ± 0.0	87.7 ± 0.8	67.2 ± 1.5	23.4 ± 1.4
sd_replay	9.4 ± 1.0	99.6 ± 0.1	91.2 ± 1.1	62.2 ± 5.1	16.3 ± 1.6
si_sd	8.7 ± 0.5	99.3 ± 0.1	91.5 ± 0.6	58.9 ± 3.2	16.4 ± 1.1
replay	7.5 ± 2.7	99.4 ± 0.2	92.9 ± 2.6	52.2 ± 15.8	13.1 ± 4.8
olora	5.5 ± 0.2	99.8 ± 0.1	95.2 ± 0.3	30.2 ± 0.3	8.3 ± 0.1
online_ewc	4.5 ± 0.5	89.2 ± 0.2	85.6 ± 0.8	25.7 ± 0.5	7.5 ± 0.6
osrm	3.4 ± 0.7	98.6 ± 0.1	96.1 ± 0.6	26.4 ± 4.7	6.2 ± 1.2
merge	3.0 ± 0.3	98.7 ± 0.0	96.7 ± 0.4	25.6 ± 2.1	5.6 ± 0.6
sd	2.8 ± 0.0	99.7 ± 0.1	97.8 ± 0.2	24.4 ± 0.3	5.2 ± 0.1
si_merge	2.5 ± 0.0	97.5 ± 0.1	95.9 ± 0.1	20.0 ± 0.8	4.5 ± 0.0
si	1.8 ± 0.1	95.3 ± 0.1	94.4 ± 0.2	16.0 ± 0.1	3.4 ± 0.1
vanilla	1.4 ± 0.0	96.0 ± 0.7	95.6 ± 0.6	12.6 ± 0.1	2.6 ± 0.0

Table 9: All configurations on LLM-QA, mean ± standard deviation over three seeds. $W(k)$ is the final-checkpoint accuracy averaged over the k most recently trained tasks.

configuration	Final	Diag	Forget	$W(10)$	$W(50)$
si_replay_olora	56.4 ± 13.2	98.5 ± 0.9	42.7 ± 14.1	87.5 ± 3.2	70.5 ± 13.8
si_replay_merge	54.8 ± 10.4	99.9 ± 0.1	45.5 ± 10.5	97.4 ± 0.5	77.9 ± 9.7
replay_merge	48.2 ± 2.8	99.9 ± 0.0	52.3 ± 2.8	98.1 ± 0.4	75.4 ± 1.9
si_sd_replay_merge	44.3 ± 8.7	99.5 ± 0.0	55.8 ± 8.7	95.5 ± 1.9	72.7 ± 11.0
si_replay_osrm	43.2 ± 2.7	99.9 ± 0.0	57.3 ± 2.8	95.6 ± 0.5	70.7 ± 4.4
sd_replay_merge	39.0 ± 1.9	99.5 ± 0.2	61.1 ± 2.0	96.5 ± 0.6	67.9 ± 1.9
si_sd_merge	31.6 ± 0.6	99.9 ± 0.1	69.1 ± 0.5	94.3 ± 0.3	57.7 ± 0.5
sd_merge	20.2 ± 0.4	99.9 ± 0.0	80.6 ± 0.4	90.5 ± 0.6	38.8 ± 0.8
si_replay	19.8 ± 0.9	99.9 ± 0.0	81.0 ± 0.9	87.5 ± 2.4	36.5 ± 1.2
si_sd_replay	15.7 ± 2.6	99.7 ± 0.0	84.9 ± 2.6	83.3 ± 3.1	30.4 ± 5.0
si_sd	14.2 ± 0.6	100.0 ± 0.0	86.6 ± 0.6	81.0 ± 1.0	27.5 ± 1.1
olora	13.9 ± 0.4	99.9 ± 0.0	86.9 ± 0.4	46.2 ± 1.9	17.3 ± 0.3
replay	12.5 ± 1.4	100.0 ± 0.0	88.3 ± 1.4	78.5 ± 4.7	23.9 ± 2.7
sd_replay	9.5 ± 0.8	99.7 ± 0.0	91.1 ± 0.8	70.1 ± 2.7	18.5 ± 1.4
sd	6.6 ± 0.1	100.0 ± 0.0	94.3 ± 0.1	56.1 ± 1.4	12.8 ± 0.1
si_merge	6.5 ± 0.4	99.9 ± 0.1	94.3 ± 0.4	53.2 ± 4.9	12.5 ± 0.9
si	4.7 ± 0.2	99.3 ± 0.1	95.6 ± 0.3	40.6 ± 3.1	9.1 ± 0.6
online_ewc	4.5 ± 0.3	99.0 ± 0.2	95.4 ± 0.2	35.7 ± 4.1	8.6 ± 0.7
osrm	4.4 ± 0.4	99.8 ± 0.1	96.3 ± 0.4	39.1 ± 2.7	8.5 ± 0.6
merge	4.0 ± 0.4	99.8 ± 0.1	96.8 ± 0.4	34.9 ± 4.4	7.7 ± 0.9
vanilla	1.3 ± 0.2	97.9 ± 1.6	97.5 ± 1.5	12.9 ± 1.4	2.6 ± 0.3

Table 10: All configurations on REAL-QA, mean ± standard deviation over three seeds. $W(k)$ is the final-checkpoint accuracy averaged over the k most recently trained tasks.

A.5.3 COMPLETE FACTORIAL TABLES

term	effect (percentage)	% var	F	p
replay	+9.49	44.3	161.1	$< 10^{-4}$
merge	+5.87	16.9	61.5	$< 10^{-4}$
sd	+5.66	15.8	57.3	$< 10^{-4}$
replay × merge	+3.57	6.3	22.7	$< 10^{-4}$
si × merge	-3.04	4.6	16.6	0.000287
si × sd × merge	-1.84	1.7	6.1	0.0192
sd × replay × merge	-1.31	0.8	3.0	0.0904
si × replay × merge	-0.79	0.3	1.1	0.297
si × replay	+0.72	0.3	0.9	0.341
sd × merge	+0.56	0.2	0.6	0.462
si	+0.34	0.1	0.2	0.649
si × sd	-0.28	0.0	0.1	0.712
si × sd × replay × merge	+0.17	0.0	0.1	0.822
sd × replay	-0.16	0.0	0.0	0.837
si × sd × replay	+0.09	0.0	0.0	0.902

Table 11: Complete saturated 2^4 factorial on SYMBOL-QA, $N = 48$, error df 32, RMSE 2.59, 91% of variance explained. The standard error of any effect is 0.75 percentage.

term	effect (percentage)	% var	F	p
replay	+18.49	43.9	341.8	$< 10^{-4}$
merge	+14.91	28.6	222.3	$< 10^{-4}$
replay × merge	+9.44	11.4	89.0	$< 10^{-4}$
si	+5.76	4.3	33.2	$< 10^{-4}$
sd	+5.02	3.2	25.2	$< 10^{-4}$
sd × replay	-3.45	1.5	11.9	0.00159
si × replay	+2.93	1.1	8.6	0.00624
sd × replay × merge	-2.62	0.9	6.9	0.0134
sd × merge	+1.69	0.4	2.9	0.101
si × sd	+1.59	0.3	2.5	0.121
si × sd × replay	-1.29	0.2	1.7	0.205
si × sd × replay × merge	-0.23	0.0	0.1	0.823
si × replay × merge	+0.20	0.0	0.0	0.845
si × merge	-0.13	0.0	0.0	0.899
si × sd × merge	-0.10	0.0	0.0	0.919

Table 12: Complete saturated 2^4 factorial on LLM-QA, $N = 48$, error df 32, RMSE 3.46, 96% of variance explained. The standard error of any effect is 1.00.

term	effect (percentage)	% var	F	p
merge	+20.54	36.4	393.4	$< 10^{-4}$
replay	+19.34	32.3	348.9	$< 10^{-4}$
replay × merge	+11.68	11.8	127.1	$< 10^{-4}$
sd × replay	-10.32	9.2	99.2	$< 10^{-4}$
si	+6.30	3.4	37.0	$< 10^{-4}$
sd × replay × merge	-4.87	2.0	22.1	$< 10^{-4}$
sd	+3.66	1.2	12.5	0.00128
si × sd × replay	-1.93	0.3	3.5	0.0711
sd × merge	+1.74	0.3	2.8	0.103
si × sd	+1.33	0.2	1.6	0.208
si × sd × replay × merge	-0.62	0.0	0.4	0.553
si × replay × merge	-0.57	0.0	0.3	0.588
si × sd × merge	+0.54	0.0	0.3	0.605
si × merge	+0.17	0.0	0.0	0.868
si × replay	+0.06	0.0	0.0	0.95

Table 13: Complete saturated 2^4 factorial on REAL-QA, $N = 48$, error df 32, RMSE 3.59, 97% of variance explained. The standard error of any effect is 1.04.

A.5.4 GREEDY COMPOSITION PATHS

Figure 5 presents a greedy nested path through the factorial for each dataset. Starting from naive fine-tuning, each step adds the remaining mechanism that gives the highest final retention when combined with the mechanisms already selected. Every method is trained independently, so the path summarizes comparisons within the factorial rather than a sequence of training stages.

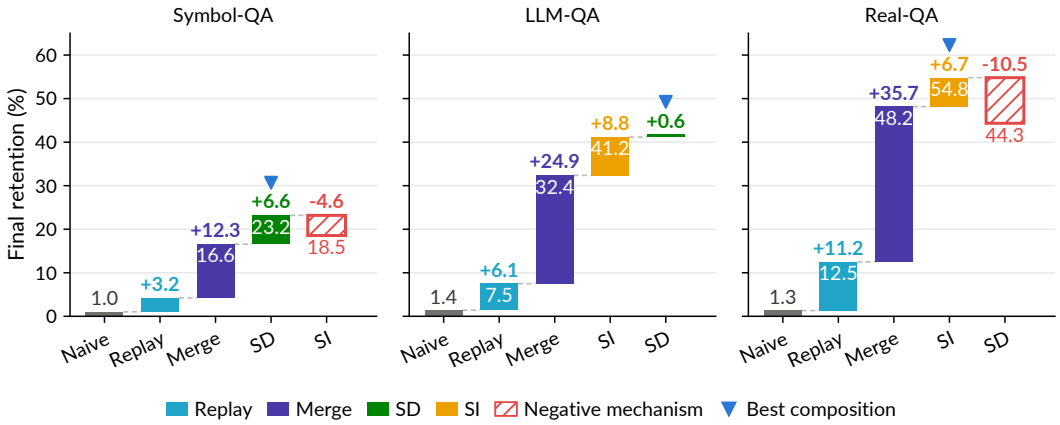

Figure 5: Greedy buildup paths through the 2^4 factorial. Each step adds the remaining mechanism with the highest final retention under the current composition. Triangles mark the strongest composition along each path.

A.5.5 RETENTION MATRICES

Figure 6 shows the retention matrix from which all reported metrics are computed. Each panel is one temporal accuracy matrix $M_{i,j}$: row i is the checkpoint after task i, column j is the task being evaluated, and only the lower triangle is observable because a task cannot be evaluated before it is trained. The diagonal represents the performance on the current training task, and everything below it is what remains of a memory as later tasks arrive. Read Figure 6 left to right, the bright band below the diagonal widens as mechanisms are added: naive fine-tuning retains only a narrow strip along the diagonal, and the leading composition keeps a broad region alive for tens of tasks. Read down a column, the same recipe covers progressively more of the triangle as the data becomes more natural. Two features of the main text are directly visible here. The band has a soft outer edge rather than a hard boundary, which is the graded decay that the survival curves quantify. The configurations

form a nested sequence, but adding another anchor does not always improve retention. Adding the weight anchor lowers retention on SYMBOL-QA, while adding the function anchor lowers retention on REAL-QA.

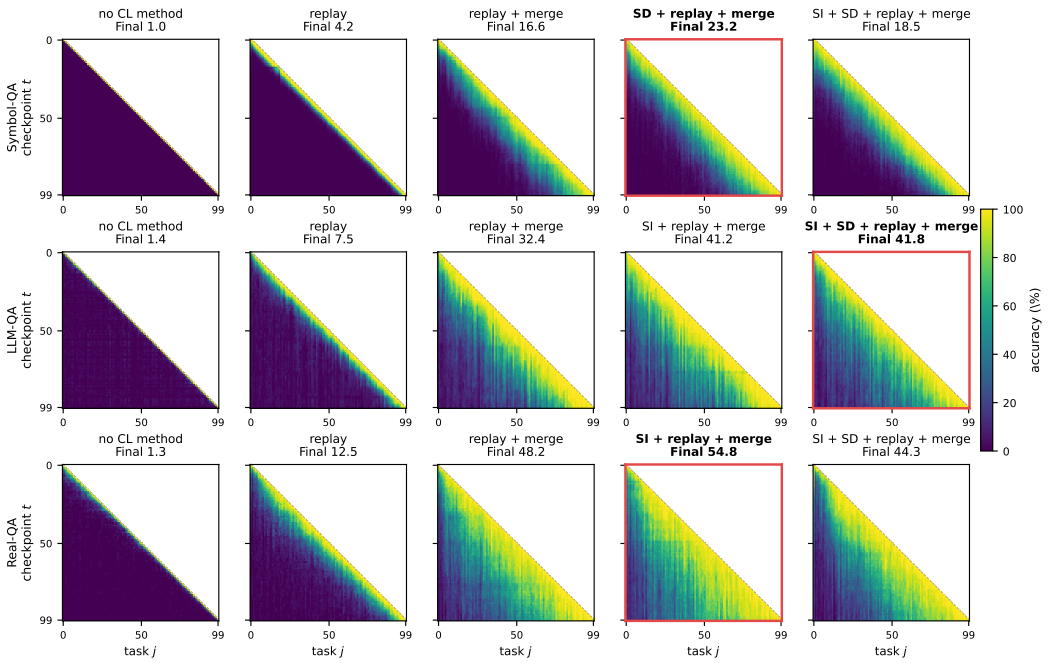

Figure 6: Temporal accuracy matrices $M_{i,j}$ along the composition chain, one row per dataset. Each column adds one mechanism and never removes one. The panel outlined in red is the strongest configuration in that row. Panels show the median seed of three, so no panel is a favorable draw.

A.5.6 TASK-GROWING LOW-RANK ALLOCATION EXPERIMENTS

The factorial compares shared LoRA and merged LoRA, two low-rank allocation rules that keep learner state independent of the horizon. To compare them with task-growing low-rank allocation rules, each dataset also uses its winning composition with merged LoRA replaced by OSRM or O-LoRA, holding every anchor and hyperparameter fixed. These cells are the only ones in the study whose retained state size grows with the number of tasks, and they are reported separately for that reason.

Additional task-growing state does not provide a consistent retention gain. Relative to the matched merged LoRA composition, O-LoRA changes final retention by -3.3, $+1.0$, and $+1.6$ points on SYMBOL-QA, LLM-QA, and REAL-QA. Sequential OSRM changes it by -0.8, -11.5, and -11.6 points. O-LoRA gives the highest mean retention on the two natural-language datasets, but its advantage over the best bounded composition is only 0.4 and 1.6 points, which is within seed variation. Section A.5.9 shows that the low-rank allocation rule has a larger effect on general capability.

$O(T)$ low-rank allocation	SYMBOL-QA	LLM-QA	REAL-QA
O-LoRA	-3.3%	$+1.0\%$	$+1.6\%$
Sequential OSRM	-0.8%	-11.5%	-11.6%

Table 14: Change in final retention after replacing merged LoRA with O-LoRA or sequential OSRM in the composition selected for each dataset.

A.5.7 COMPOSITIONS THAT FAIL THE ACQUISITION GATE

Two cells fail to memorize rather than fail to retain, and both compose a weight anchor with the fold: `si_sd_merge` reaches Final $= 6.8 \pm 5.9$ with Diag $= 95.0 \pm 8.0$, and `online_ewc` reaches

1.0 ± 0.6 with $\mathrm{Diag} = 86.0 \pm 15.1$. The large standard deviations are the signature: on some seeds the accuracy-matrix diagonal collapses outright while on others training proceeds normally, so the mean describes a mixture of two behaviours rather than a typical run. Every other configuration in the study holds Diag between 96 and 100.

This is the failure mode the compatibility rule predicts. A quadratic penalty is defined in the coordinates in which its importance estimates were measured; merged LoRA folds the adapter into the base weights and re-initializes, so those coordinates no longer denote the same function, and the penalty resists motion the model needs in order to fit the current task. The consequence is a loss of plasticity, which is why it appears on the diagonal rather than in the off-diagonal decay.

A.5.8 SEED VARIANCE AND THE GENERATION-TEMPERATURE CONTROL

Seed standard deviations partition cleanly by whether a configuration carries the data anchor. Replay-bearing stacks give 2 to 6 points (`replay_merge` ± 5.8, `sd_replay_merge` ± 4.2), while configurations without replay are tight (`sd_merge` ± 1.3, `sd` ± 0.0, singletons ± 0.1 to ± 0.2).

A controlled comparison isolates the mechanism. Re-running one composition at replay generation temperature 0.7 rather than 1.5, with three seeds and all else fixed, gives 21.2 ± 7.9 against 20.8 ± 2.1. The means are statistically indistinguishable while the seed spread differs roughly fourfold, so generation diversity acts on the variance of the outcome and not on its expectation. Sharp, low-entropy replay concentrates the pseudo-data on few modes, and whether those modes align with a given seed's trajectory determines the run. This is also why single-seed probes are unsafe here: the low-temperature arm contains a seed reaching 30.3 alongside seeds at 16.2 and 17.1, and reporting the first alone would have suggested a large improvement where there is none.

Two byproducts of that re-run are worth recording. The outlying seed reproduced the same value across a three-week interval of repository changes, which is direct evidence that training is deterministic given seed and hyperparameters. And the same winner's-curse mechanism that operates over configurations (Section A.5.10) operates over hyperparameters.

A.5.9 HELD-OUT GENERAL CAPABILITY EVALUATION

We evaluate the final checkpoints and the unmodified base model on GSM8K (Cobbe et al., 2021), MATH (Hendrycks et al., 2021), MGSM (Shi et al., 2022), and MMLU-Redux (Gema et al., 2025). The base model scores 83.6%, 57.6%, 67.5%, and 68.7%, respectively, with an unweighted average of 69.4%. We average trained-model scores over three seeds. GSM8K, MATH, and MGSM are scored by extracting and checking answers from generated text.

For MMLU-Redux, we select an answer directly from the model's probabilities rather than generating text. Each prompt contains two fixed examples followed by the question and four choices labeled A through D, and ends with `Answer:`. We compare the next-token log probabilities for the four answer labels and select the label with the highest score. The evaluator uses the first token ID obtained by tokenizing each letter with a preceding space. This *letter log-likelihood* score is the logarithm of the probability assigned to a candidate answer-label token, not to the full choice text. Every question receives a prediction without requiring the model to generate a valid answer letter. The scoring therefore avoids answer-extraction failures, although performance can still depend on the prompt.

Figure 4 compares general capability across low-rank allocation rules. O-LoRA changes final retention only slightly relative to merged LoRA, but the difference in general capability is larger on the two natural-language datasets. Averaged across the four benchmarks, O-LoRA achieves 26.8% accuracy after LLM-QA and 28.8% after REAL-QA. Merged LoRA reaches 13.0% and 13.4%, while sequential OSRM reaches 8.2% and 8.0%. The O-LoRA compositions also achieve 45.8% and 52.0% on MMLU-Redux, respectively. Even so, their four-benchmark averages remain more than 40 percentage points below the base model.

After SYMBOL-QA, all three allocation rules in Figure 4 reduce average general capability accuracy to below 8.0%. Some GSM8K generations repeat fragments of the training format instead of providing mathematical answers. However, output generation failures do not fully explain the low scores. MMLU-Redux accuracy remains near the 25% chance level even when answer labels are selected directly from their log probabilities.

O-LoRA leaves the base parameters unchanged and accumulates one adapter per task. Removing these adapters restores the original model, but also removes the learned task updates. Merged LoRA and sequential OSRM instead incorporate each update into the base parameters, so the original model cannot be recovered simply by detaching an adapter. This recoverability is distinct from preserving general capability with the learned updates active. Our comparison does not isolate whether this structural difference causes O-LoRA's higher general capability scores. Both O-LoRA and sequential OSRM retain state that grows linearly with the number of tasks, and neither prevents substantial general capability loss after 100 tasks.

A.5.10 SELECTION VERSUS EVALUATION

The search of Section 4.3 and the factorial of Section 5 disagree about which configuration is best on SYMBOL-QA: the search selects `si_sd_replay_merge` and the final evaluation ranks `sd_replay_merge` above it. The two were separated by 1.0 point on the development order, which is inside the seed spread of either.

This is the expected behaviour of an argmax over many noisy candidates. The selected configuration regresses from 23.4 to 18.5 on the report order, while the eventual leader is stable at 22.4 and 23.2. Because SYMBOL-QA tasks are exchangeable by construction, the development and default orders are draws from the same distribution, so the movement is seed noise rather than overfitting to a particular ordering. The design anticipates this: the search selects a family under a budget, and the factorial provides unbiased estimates and the decomposition. It is also why the paper reports a recipe and a set of effects rather than crowning a single configuration.

A.5.11 CONTROLS

Forward transfer is 0.0 ± 0.0 on SYMBOL-QA and below 3.1 points elsewhere, as expected when a task is unseen before its turn; the small positive values on the natural-language datasets reflect shared surface form rather than knowledge of the specific facts. The `\boxed{}` emission rate is 100% on every configuration and both at the diagonal and at the final checkpoint, so no configuration loses measured accuracy through format failure, and content retention is separated cleanly from format retention.