BUDGETED CACHE REPAIR FOR CROSS-CONTEXT KV-CACHE REUSE

Anonymous authors
Paper under double-blind review

ABSTRACT

Cross-context KV-cache reuse predicts a shared segment's keys and values under a new prefix instead of recomputing them, and has been reported to do so without quality loss. We find otherwise, and identify two problems. (1) **A hidden cost**: on MMLU and GSM8K, reuse costs substantial accuracy. (2) **A decision at the wrong unit**: no rule for deciding *whether* to reuse a cache removes that cost. What does help is choosing *which parts* of the cache to recompute, and the value of choosing well falls as the unit of choice grows: informed selection removes 49.5% of the cache error beyond chance at single rows (one token's keys and values), 10.6% at 64-token chunks, and nothing at the level of whole calls. **B**udgeted **C**ache **R**epair (**BCR**) acts at the unit where selection still pays. It drafts two tokens from the assembled cache, ranks cache rows by the attention those tokens pay them, and recomputes a fixed budget of rows exactly, in one of three layouts. The cost is paid rather than predicted away, and *the draft that fails as a gate succeeds as a selector*. BCR restores GSM8K to dense-prefill accuracy while still serving most calls from cache, and its best layout outperforms every reuse baseline's mean in the reference grid. The draft also beats a coin-flip selector at the same budget—a control prior evaluations lack.

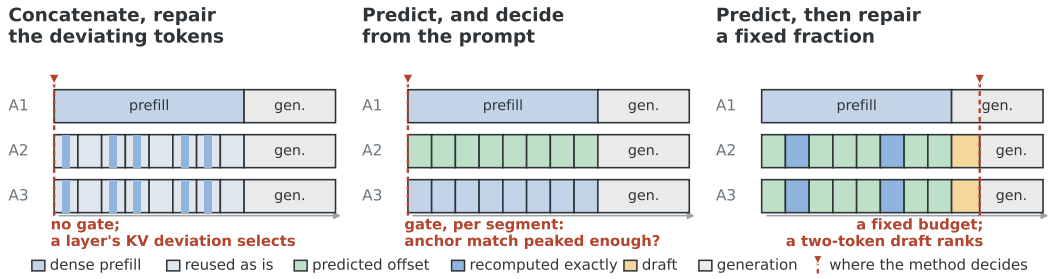
1 INTRODUCTION

A multi-agent LLM system processes the same context again and again. When three agents and an aggregator each receive the same question and each other's outputs, the question is prefilled four times and the first answer three times. This redundancy motivates cross-context KV-cache reuse (Kwon et al., 2023; Gim et al., 2024; Yao et al., 2025): rather than recomputing a segment's keys and values under every new prefix, a reuse method estimates whether the representations computed earlier are still valid. KVCOMM (Ye et al., 2025) does this with an online pool of *anchors*—segments stored together with the deviation a prefix change induced in them—and reports over 70% cache reuse and up to $7.8\times$ faster time-to-first-token with no measurable quality loss.

A reused cache is not always safe to serve, so every reuse system needs a rule for deciding when its prediction can be trusted. CacheBlend (Yao et al., 2025), Figure 1(a), sidesteps the question by recomputing selected tokens instead of trusting the seams. KVCOMM instead applies a *gate*: once per reused segment, before generation begins, it thresholds the entropy of the segment's similarity to the stored anchors (§3). The gate's defining limitation, Figure 1(b), is that *it decides before any information about the answer exists*: it can inspect only the prompt and the predicted cache, so it uses the cache's geometry as a surrogate for whether reuse will change the answer.

We find that this prompt-side surrogate is poorly aligned with the errors reuse actually causes. Reuse costs substantial accuracy on two of three benchmarks, and the cost survives a check for anchor-pool warm-up. Tightening the gate does not remove it, and neither does choosing which calls to reuse: across five call-level allocation policies, including a random baseline, accuracy tracks the amount of reuse rather than the policy. Nor does deferring the decision until generation has begun, the fix that cache eviction uses (Li et al., 2024; Zhang et al., 2023; Wang et al., 2025): our generation-time gate OPENGATE, which decides from two draft tokens, loses 6.7 points to DENSE on 300 matched GSM8K problems (exact McNemar $p = 0.009$ on the problems where the two disagree; §4 gives the test form). Together these results say the problem is not *when* or *whether* to reuse but *what unit* the

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.



(a) **CacheBlend.** No offset; a layer's deviation selects.

(b) **KVCOMM.** Offset from anchors; a gate refuses A3.

(c) **BCR (ours).** Offset kept; a draft selects a budget.

Figure 1: **Three ways to serve a shared segment.** Rows show three agents sharing one segment in execution order: A1 is the *first caller* and computes it exactly; A2 and A3 are *later callers* and can reuse it from cache. The horizontal axis denotes token position.

decision controls: a whole-call gate can only trust or reject an entire cache, even when the error sits in a small subset of its rows.

This motivates a different approach: *repair part of the cache instead of trusting all of it*. **B**udgeted **C**ache **R**epair (**BCR**) uses generation itself to find the parts of a reused cache most worth repairing. It drafts two tokens from the assembled cache, ranks cache rows by how much attention those tokens pay them, and recomputes exactly a fixed budget of the highest-ranked rows. We study three repair layouts, illustrated in Figure 1(c): fixed chunks, grown spans, and individual rows (§3). BCR pays the hidden cost directly, and pays it at a unit where informed selection can still reduce it.

Our contributions are as follows.

- **Budgeted Cache Repair.** We introduce BCR, a generation-guided method that uses a two-token draft to select, and then exactly recompute, a fixed budget of KV rows, in three repair layouts: fixed chunks, grown spans, and individual rows.
- **Accuracy recovery under reuse.** BCR recovers most of the accuracy reuse costs, restoring GSM8K to dense-prefill accuracy while keeping most of the reuse, and on each benchmark of the reference grid its best layout leads every reuse baseline's mean, KVCOMM and CacheBlend alike. Recovery is task-dependent: gains on MMLU are limited, and HumanEval has little degradation to recover.
- **What determines the error of reuse.** The value of informed selection falls as the unit of decision grows, from single rows to chunks to whole calls, and the cost itself is workload- or backbone-specific.

2 Related Work

Reuse across contexts. Prefix caching (Kwon et al., 2023) reuses a cache only under an exact prefix match. Prompt Cache (Gim et al., 2024) extends reuse to position-agnostic segments, and CacheBlend (Yao et al., 2025), Figure 1(a), repairs independently cached chunks by recomputing selected tokens. KVCOMM (Ye et al., 2025), Figure 1(b), takes a different route: it predicts the deviation a prefix change causes, using an online anchor pool, and decides per segment whether to reuse the cache. We study this formulation and ask whether such prediction preserves accuracy, including after the anchor pool has warmed up. Multi-agent pipelines (Du et al., 2024; Zhuge et al., 2024) make such reuse frequent; AgentPrune (Zhang et al., 2025) attacks the same cost from the communication side. Cache-compression methods face a related choice of what to keep, using positional rules (Xiao et al., 2024), reconstruction (Zweiger et al., 2026; Kim et al., 2025), or attention-based scores (Li et al., 2024; Zhang et al., 2023; Cai et al., 2025; Feng et al., 2025); recent work shows that such input-side criteria can be misaligned with task performance (Wang et al., 2025). BCR differs from all of these in what it decides: it fixes a repair budget and chooses *what to repair*, rather than whether to trust the cache as a whole. Like CacheBlend it pays for selective recomputation, but it applies the idea

to cross-context reuse and studies how the unit of repair changes its value; Figure 1(c) draws the contrast.

Selecting what to recompute. Recent work improves selective recomputation by ranking tokens with attention from the user query (Teng et al., 2026; Wang et al., 2026) or from a small model that approximates the target (Yang et al., 2025), and reports substantial gains at matched budgets. Draft passes that generate tokens before committing (Leviathan et al., 2023; Wang et al., 2025) likewise motivate generation-time signals for selection. These methods ask *which tokens to select* within a fixed formulation. We isolate two factors instead: what informed selection is worth relative to an uninformed baseline, and how that worth depends on the *unit* of selection. We therefore compare informed and random selection at the same repair budget, and vary the unit from whole calls to chunks to individual cache rows. That controlled view is the focus of BCR.

3 Budgeted Cache Repair

The reuse problem. A request runs on a directed acyclic graph of N agents. Agent i's prompt is assembled from a private system prompt S_i, the shared task T, and the outputs of its predecessors, so in the fully connected configuration studied here a segment such as T is prefilled $N{+}1$ times. Let $C(x \mid P)$ denote the KV cache of a token span x computed under prefix P, one *row* per token. Every entry of that cache depends on P through attention, so exact reuse requires an identical prefix—the condition prefix caching assumes (Kwon et al., 2023). KVCOMM (Ye et al., 2025) relaxes it by *predicting* the dependence. For a segment x already seen under some prefix P, it stores an *anchor*: the deviation

$$\Delta \;=\; C(x \mid P) \;-\; C(x \mid \emptyset), \tag{1}$$

between the segment's cache under P and its cache under no prefix at all. At serve time, a new segment's deviation is approximated by a similarity-weighted combination of stored Δ's, added elementwise to the cache.

The gate. An agent's prompt contains several such segments—the task, each predecessor's output—and each is decided separately: trust the approximation (*reuse*) or pay a full prefill for that segment (*dense*). KVCOMM decides *before generating any token*, from an input-side statistic. Let d_j be the L_2 distance in value space between the candidate segment and anchor j, collect the distances over the n available anchors as $d = (d_1, \ldots, d_n)$, and let $s = \mathrm{softmax}(-d)$. The rule thresholds the entropy of s:

$$\text{reuse} \quad\Longleftrightarrow\quad H(s) \;\le\; \gamma \log_2 n, \qquad H(s) = -\textstyle\sum_j s_j \log_2 s_j, \tag{2}$$

with a hard fallback to dense whenever $n \le 1$. A peaked s means the candidate closely matches one stored anchor, whose deviation should transfer. Since $\log_2 n$ is the largest entropy s can have, the threshold γ (default 0.3) is the fraction of maximum ambiguity the rule tolerates, and $\gamma \to 1$ recovers unconditional reuse. Equation (2) is the object this paper examines.

The repair. Predicting a cache instead of computing it costs accuracy, and no setting of the gate avoids that cost; §4.1 measures both. Reusing fewer caches, and choosing which, does not avoid it either: §4.2 rules that out and prices what choosing well is worth at each unit. What remains is to repair part of the reused cache rather than decide whether to trust all of it. As Figure 1(c) illustrates, BCR recomputes a fixed *budget* ρ of the cache's rows exactly and reuses the rest. Whenever KVCOMM serves a call from an assembled cache, BCR generates two tokens, rolls the draft back, and scores every cached row by the attention the draft paid it. Let Q^ℓ be the two query rows kept at layer ℓ, indexed by step t and head h, and $\widetilde{K}^\ell$ the predicted keys. Row j scores $a_j = \sum_{\ell>1} \mathrm{mean}_{t,h} \left[\mathrm{softmax}(Q^\ell_{t,h} \widetilde{K}^{\ell\top}_h / \sqrt{d_{\text{head}}})\right]_j$, and a group of rows scores $\bar{a}_i = \mathrm{mean}_{j \in i}\, a_j$. The highest-scoring rows or groups, up to a budget of ρ of the cached rows, are recomputed exactly before generation begins. The keys scored are the predicted ones: the signal that decides what to fix is read from the tensors under suspicion. The draft is the same deferral OPENGATE applies to *whether* to reuse, applied instead to *which rows* to repair; App. B gives the procedure as Algorithm 2.

The layouts. Spending a budget at all recovers 7.9 of the 13.7 accuracy points reuse costs on GSM8K, and the rule that places the budget recovers 5.4 more (§4.3). How large the budget should

Table 1: **The budget: how large, in what layout, at what price.** (a) two agents, 150 matched GSM8K problems, 64-token chunks drawn by coin at four seeds so ρ is the only variable; *mean* and *spread* are the accuracy over those seeds and its range, $\max - \min$. KVCOMM is ρ=0, DENSE ρ=1. (b) rows recomputed per agent call at four agents. (c) time to the first token on a 4096-symbol synthetic segment at ρ=0.20, $\times$ the speedup against DENSE. Full tables in Tables 10, 11 and 15.

ρ	mean	spread
0.05	76.5	3.3
0.20	77.5	11.3
0.40	78.8	2.7
KVCOMM	77.3	–
DENSE	82.7	–

(a) The budget swept, four coin seeds

Method	MMLU	GSM8K	HEval
BCR-chunk	159	211	138
BCR-grow	174	221	152
BCR-single	167	216	154
CacheBlend, reuse	417	474	97
CacheBlend, cost	158	206	132

(b) Rows recomputed per call, four agents

Method	ms	$\times$
DENSE	542.7	–
KVCOMM	72.9	7.63
BCR-chunk	190.1	2.85

(c) Time to first token, 4096 symbols

be, Table 1(a)'s coin-drawn sweep cannot resolve: at ρ=0.20 its seeds span 11.3 accuracy points, wider than the $82.7 - 77.3$ gap they bracket. What it leaves open is the layout the budget is spent in, and we test three, coarsest first:

- **BCR-chunk** divides the cache into fixed 64-token chunks and recomputes the highest-scoring ρ of them. Spans have a fixed length and sit on a grid.
- **BCR-grow** drops the fixed unit: it seeds on the best-scoring row and extends the span outward while a neighbour scores at least τ of the seed. Lengths follow the score surface, and positions are not aligned to anything.
- **BCR-single** drops grouping altogether and spends the budget one row at a time, the finest unit the budget admits.

One budget, three units. Unqualified, BCR means all three. Table 1(b) puts their row counts within 12% of the smallest (154 against 138 on HumanEval, the widest of the three; 9.4% on MMLU and 4.7% on GSM8K), so the three share one budget and differ in the unit they spend it in; comparing them isolates the unit, not the spend. Table 1(c) prices the budget in time: 190.1 ms to the first token, against reuse's 72.9 and DENSE's 542.7.

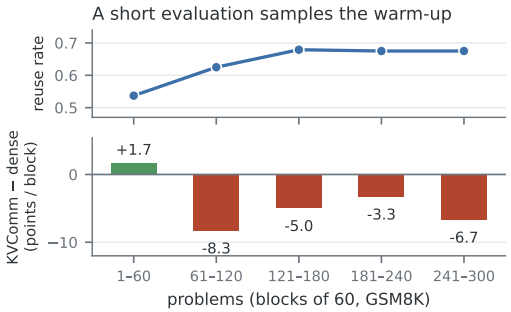
4 EXPERIMENTS

The measurement setting. GSM8K (Cobbe et al., 2021) and MMLU (Hendrycks et al., 2021) run on Llama-3.1-8B-Instruct (Grattafiori et al., 2024) and are scored by exact match and letter matching respectively; HumanEval (Chen et al., 2021) runs on Qwen2.5-Coder-7B-Instruct (Hui et al., 2024) and is scored by unit tests. Two configurations recur, and every result is labelled with the one it uses. The *reference* configuration follows the released implementation: roles cycle rather than stay pinned, and the shipped question budgets—150, 153 and 160 items—are run at 2 to 5 agents in Table 3. Its three-`MathSolver` GSM8K cell, with $N{+}1 = 4$ calls per request, is where §4.1 begins: 300 problems, the first 60 of which also serve to sweep γ. The *pinned* configuration assigns five agents fixed roles, over 300 matched GSM8K and MMLU problems in Table 5 or the 150 problems of Table 4(a); pinning concentrates divergence at the entry node, where §4.1 measures it. Statistical significance in this paper comes from these single runs over distinct items, never from Table 3. Decoding is greedy throughout, with 512 maximum new tokens, on one NVIDIA A100 80GB; anchor-pool hyperparameters stay at the repository defaults. App. G.1 gives the full environment and scoring rules.

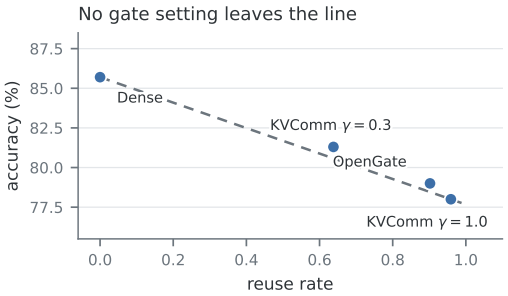
The statistical protocol. Within each comparison, every condition runs the *same* problems with the same decoding parameters, so every comparison is paired. We report the *discordant* split b/c before any p-value: b counts problems solved only by the first-named condition, c those solved only by the other, unless a table's caption fixes the orientation for its rows. An exact McNemar test on these paired disagreements then asks whether they favour one condition more often than chance would. The split is more informative than marginal accuracy when two conditions differ on only a handful of problems. Results are reported per agent count rather than pooled across the grid: the four cells reuse

Table 2: **What the gate can and cannot do.** The first 300 GSM8K problems in the reference configuration. b counts problems DENSE solves and the row misses, c the reverse, *disc.* their sum; * marks $p < 0.05$. *cost* is points against DENSE, taken from the discordant counts rather than the rounded accuracies. *rate* is the fraction of the 1200 agent calls served without a dense prefill.

		Accuracy vs. DENSE						Reuse
Method	reuse when	acc. (%)	95% CI	cost	b/c	disc.	p	rate
DENSE	never	85.7	[81.2, 89.2]	–	–	–	–	0.000
KVCOMM	$H(s) \leq \gamma$=0.3 $\log_2 n$	81.3	[76.5, 85.3]	−4.3	23/10	33	0.035*	0.638
OPENGATE	conf. $\geq \theta$=0.5	79.0	[74.0, 83.2]	−6.7	37/17	54	0.009*	0.902
KVCOMM	$H(s) \leq \gamma$=1.0 $\log_2 n$	78.0	[73.0, 82.3]	−7.7	35/12	47	0.001*	0.959



(a) **The anchor-pool warm-up.** The pool fills as the run goes on, so the first block reuses least and is the only one where reuse does not lose.



(b) **The gate sweep.** Every setting, and the answer-side gate, lands on the line DENSE and unconditional reuse define.

Figure 2: **What reuse costs, and what the gate can do about it.** GSM8K, n=300 (Tables 2 and 6).

the same problems, so pooling their pairs would treat repeated measurements as independent and inflate the effective sample size fourfold. Pooled splits therefore appear only descriptively, never as tests; App. H.1 records the ones we withdrew.

4.1 WHAT REUSE COSTS

We measure two things in order: whether predicting the cache instead of recomputing it costs accuracy, and whether that cost can be blamed on the *rule* that decides when to trust a prediction—the gate of Equation (2). Both measurements compare the same problems solved two ways, under the matched-pair protocol above.

The cost of reuse. Table 2 puts the cost at −4.3 points on 300 matched GSM8K problems in the reference configuration ($p = 0.035$ on 33 discordant pairs); pinned, the cost is 13.7 (§4.3). Unconditional reuse loses 7.7 ($p = 0.001$), and OPENGATE, the answer-side gate of App. A, loses 6.7 ($p = 0.009$). No setting of the gate, and no replacement we built for it, avoids the cost of predicting the cache; recomputing part of the prediction does (§3), while changing the decision about the whole cache does not. At $\gamma = 0.3$, reuse loses 23 problems that DENSE solves but *gains* 10 that DENSE misses, Table 2's 23/10: 33 problems change outcome for a net change of only 13, or 4.3 points. The approximated caches produce different answers, not occasional slips.

The warm-up effect. On the first 60 problems—a plausible pilot—reuse is *ahead* by 1.7 points, the opposite of the cost just measured. Figure 2(a) resolves this block by block, and the pattern follows the method's own dynamics rather than noise. The first block has the run's lowest reuse rate, 0.537, against 0.625 to 0.679 in later blocks, because a dense pass is what *materializes* an anchor: the realized deviation of Equation (1) needs the exact cache for its subtraction, so early requests build the pool rather than draw on it. Every later block reuses more, and every later block is negative. A short evaluation therefore samples mostly the regime before the cost appears.

Table 3: **What reuse costs across agent counts.** Reference configuration, γ=0.3 (1.0 on HumanEval and on GSM8K's CacheBlend rows); 153, 150 and 160 items, matched per problem. **Bold** is the best score in a column excluding DENSE, shading the best BCR mean; *avg* carries no test. DENSE reuses nothing and the cost-matched pairing matches rows, so neither has a reuse row. Splits in Tables 13 and 14.

	MMLU					GSM8K					HumanEval				
# Agents	2	3	4	5	*avg*	2	3	4	5	*avg*	2	3	4	5	*avg*
Accuracy (%), HumanEval pass@1															
DENSE	74.5	75.2	75.2	75.2	75.0	82.7	86.0	84.7	84.7	84.5	87.5	86.2	85.6	85.0	86.1
KVCOMM	71.9	73.2	72.5	72.5	72.5	77.3	82.0	80.0	84.0	80.8	86.2	85.6	85.6	85.6	85.8
CacheBlend, reuse-matched	69.3	69.3	69.3	68.6	69.1	82.0	**86.7**	84.7	81.3	83.7	88.8	85.6	85.0	86.2	86.4
CacheBlend, cost-matched	72.5	71.2	69.3	73.9	71.7	**85.3**	83.3	83.3	82.0	83.5	85.0	84.4	85.6	85.6	85.2
BCR-chunk	74.5	**73.9**	72.5	75.2	74.0	82.0	86.0	85.3	**85.3**	**84.7**	86.2	86.2	85.6	85.6	85.9
BCR-grow	**77.1**	72.5	**75.8**	**75.8**	**75.3**	80.7	82.0	81.3	80.7	81.2	**89.4**	**88.1**	**88.1**	**88.1**	**88.4**
BCR-single	71.2	71.9	73.9	73.2	72.5	84.0	83.3	**86.0**	84.7	84.5	88.8	**88.1**	86.9	86.9	87.7
Reuse (%)															
KVCOMM	45.1	47.4	47.3	46.9	46.7	64.9	60.0	55.2	52.1	58.1	89.8	87.8	85.6	84.1	86.8
CacheBlend, reuse-matched	46.0	49.5	48.5	48.9	48.2	65.6	60.3	55.6	52.2	58.4	87.9	87.2	85.4	83.9	86.1
BCR-chunk	45.5	48.4	48.6	48.0	47.6	65.6	60.3	55.6	52.2	58.4	89.6	87.5	85.7	84.4	86.8
BCR-grow	46.4	50.8	52.3	52.0	50.4	63.8	58.8	53.9	50.6	56.8	89.2	87.8	86.2	84.9	87.0
BCR-single	45.5	48.2	48.6	48.1	47.6	64.9	59.8	55.1	51.9	57.9	89.8	88.6	87.4	86.2	88.0

The gate sweep. If the loss came from misplaced trust in predictions, tightening the rule should recover quality. Table 2 sweeps it, and Table 9 sweeps it more finely on the first 60 problems, with latencies. The setting $\gamma = 1.0$ is not an extrapolation but the exact "never refuse" policy, since $H(s) \leq \log_2 n$ makes the test unfireable there—which App. G.3 uses as a check that the setting took effect. Every setting stays below DENSE, and the cost deepens as reuse rises: -4.3, -6.7 and -7.7 accuracy points, at $p = 0.035$, 0.009 and 0.001. The gate moves the operating point along the one reuse–accuracy line that Figure 2(b) draws; it does not lift the line. The loss lives in the approximation itself, not in the rule that decides when to apply it.

The agent-count grid. Table 3 repeats the comparison at two to five agents, and DENSE leads KVCOMM in all eight MMLU and GSM8K cells. *No single cell is significant* at the repository's own sample sizes: Table 4(d) gives all twelve splits; the largest, GSM8K at two agents, is $17/9$, $p = 0.169$. Pooling the counts is not a valid test (§4), so the grid gives direction, not magnitude; significance rests on the single n=300 runs.

The same grid puts the repair at DENSE's ceiling, where the reuse baselines stop short. In Table 3's means—descriptive, since the four cells re-run the same problems—**BCR-chunk** takes GSM8K at 84.7, and **BCR-grow** takes MMLU at 75.3 and HumanEval at 88.4: level with DENSE's 84.5, 75.0 and 86.1, and clear of KVCOMM's 80.8, 72.5 and 85.8 and of CacheBlend's best, 83.7, 71.7 and 86.4. The three layouts spend one budget, and which of them leads depends on the benchmark (§4.3); some layout beats KVCOMM in all twelve cells and each CacheBlend pairing in eleven.

The comparison matches conditions on budget, and Table 3's GSM8K cell re-runs to the printed scores exactly. MMLU does not. Its analysis agent splices a live, uncached `en.wikipedia.org` fetch into the prompt, so every condition—DENSE included—depends on what Wikipedia said that day. Disabling retrieval makes the pipeline exactly reproducible, and that offline regime, documented in App. G.2, is what Table 3's MMLU column now reports. Measured offline, the cost is 2.0 to 2.7 points—three or four problems of 153 in Table 4(d)'s counts, none significant. The direction is unchanged, and the caveat applies to any MMLU number measured with roles cycling, the reference implementation's included.

The third benchmark. HumanEval behaves differently: reuse costs nothing. Its KVCOMM row runs at γ=1.0, the setting whose reuse rate matches the published one, and across four cells it is level with DENSE at reuse above 84%, $28/26$ in Table 4(d)'s pooled column. The tokenization term that §4.3 prices at 6.7 points on MMLU from Table 5(a) is measured there, not here, so we do not decompose this null. The cost is thus workload- or backbone-specific: present on two benchmarks, absent on the third. Where it is present, no rule avoids it—the premise from which §3's budget starts.

Table 4: **The evidence behind the cost, the two nulls and the layout ordering.** (a) five agents, 150 matched GSM8K problems, roles pinned; *resid.* is the points earned beyond Equation (3). (b) six prompts; *gap* is the cache error informed selection removes beyond chance, as a share of the unrepaired 0.1054, *won* the prompts where it beat the coin. (c) the three layouts of §3, b/c splits pooled over four agent counts. (d) is Table 13's cost rows, p in parentheses. Throughout b counts problems KVCOMM misses and the other solves. Full tables in Tables 7, 8 and 13.

Policy	reuse	resid.
Protect-the-Root	0.456	+0.9
Random	0.454	+1.6
Protect-the-Leaf	0.504	+0.4
DENSE	0.000	+0.0
KVCOMM	0.600	+0.0

(a) Allocation policies

unit (rows)	gap %	won of 6
1	49.5%	6
64	10.6%	5
1024	−5.7%	1

(b) Unit of choice

	chunk	grow	single
GSM8K	47/24	41/39	44/22
MMLU	21/12	54/37	50/50
HumanEval	4/3	29/12	27/15

(c) Layouts against KVCOMM

DENSE vs. KVCOMM	2 agents	3 agents	4 agents	5 agents	pooled
MMLU	16/12 (0.572)	15/12 (0.701)	18/14 (0.597)	18/14 (0.597)	67/52
GSM8K	17/9 (0.169)	11/5 (0.210)	15/8 (0.210)	9/8 (1.000)	52/30
HumanEval	8/6 (0.791)	7/6 (1.000)	5/5 (1.000)	8/9 (1.000)	28/26

(d) The grid's cells, b/c (p), and their totals across the four counts

4.2 WHY A BUDGET, AND AT WHAT UNIT

The allocation policies. One might instead reuse fewer caches and choose which. Table 4(a) runs the policies of App. A on 150 GSM8K problems with roles pinned: Protect-the-Root and Protect-the-Leaf force a dense prefill at the entry node and at the deepest node, and Random gives up reuse at calls drawn by a biased coin; OPENGATE is the exception, which §4.1 measures in the reference configuration. The policies are distinguishable only by where they land on one line. Draw that line between DENSE at reuse 0 and unmodified KVCOMM at reuse 0.600, scoring 85.3 and 75.3 in Table 7; its slope is $(85.3 - 75.3)/0.600 = 16.7$:

$$\text{accuracy } (\%) \;=\; 85.3 \;-\; 16.7 \times \text{reuse rate}. \tag{3}$$

The three policies between the endpoints land on the line, none more than 1.6 points off (about half a standard error at n=150; the endpoints define the line, so only these three residuals are free). The sharpest case is Random, which consults nothing—not the anchors, not the answer, not the topology—yet reaches Protect-the-Root's reuse to within 0.002 (0.454 against 0.456) and carries the larger residual of the pair. The two structural policies land at reuse 0.456 and 0.504, and a divergence measurement says which should win: conditioned on predecessors that already matched DENSE, the entry node diverges where the deepest does not, so the structural hypothesis predicts a gap. None resolves. The measurement was right about where error originates but wrong as a prescription: what determines the outcome is how much of the context was predicted, not where in the graph. One policy that fails to separate is weak evidence; four of different kinds—cache geometry, graph position twice, and nothing at all—make it a property of the problem. The cost is a function of how much you reuse, not of which calls. That claim is narrower than it sounds, for the reason the next measurement gives.

The unit of choice. Large reported gains for choosing which *tokens* to recompute sit awkwardly beside the null above for choosing which *calls* to reuse. The two are not in conflict: the unit of choice is the variable, and choosing well loses value as the unit grows. To show this, we start from a reused cache of a 2048-token segment, restore a nominal 20% of its rows, and vary only how those rows are grouped and chosen—by attention from the consuming context's decode queries, or at random. Error is the relative L_2 distance of the attention output from a dense cache's, pooled over layers and heads because a recomputed span is shared by all of them. Table 4(b) gives the curve. Informed selection removes 49.5% of the cache error beyond chance at single rows, 10.6%—a fifth of that—at 64-token chunks, and −5.7% at 1024, where it is nominally *worse*; the prompts it wins fall from 6 to 5 to 1 of six. Table 8 carries the paired t behind those counts ($p = 0.08$ at 64) and notes that the budget drifts above the nominal 20% once chunks are coarse, so 256 and coarser are not budget-matched, pending a matched sweep. Two results that looked contradictory are thus one ordering, with our call-level null at one end and the literature's token-level gains at the other. The claim should be read with its unit

Table 5: **Where repair works, and what it is worth against a published selector.** Five agents pinned to one role, 300 matched problems. In (a) b counts problems KVCOMM solves at its shipped γ=0.3 and the row misses, c the reverse, p in parentheses, * marking $p < 0.05$; $-$ marks a row with no split of its own—KVCOMM itself, the † five-seed coin mean, and the token-matched control, which only MMLU needs. Shading marks the repair row. (b) pools the reference grid over four agent counts, descriptively (Table 14).

	GSM8K		MMLU	
Method	acc.	b/c (p)	acc.	b/c (p)
KVCOMM	71.0	–	51.7	–
BCR-chunk, coin	78.9†	–	50.0†	–
BCR-chunk, draft	84.3	9/49 (0.000*)	54.7	12/21 (0.163)
token-matched DENSE	–	–	59.3	25/48 (0.010*)
DENSE	84.7	10/51 (0.000*)	66.0	21/64 (0.000*)

(a) Cost and repair at n=300

pairing	GSM8K	MMLU	HEval
reuse	25/31	39/69	29/26
cost	23/30	47/61	21/26

(b) CacheBlend vs. **BCR-chunk**

attached: *allocation does not matter across agent calls*. The mechanism is concentration. Within a segment the attention mass is extremely uneven, so there is much to choose between; across calls, the candidates are whole prompts of the same shape.

The chunk unit is where KVCOMM's data structures can act, since its offset, anchor filter and re-rotation all assume a contiguous span, as App. A.2 details. The curve therefore predicts that at 64-token chunks a selector keeps about a fifth of its single-row advantage, and that BCR should gain from repairing at all more than from repairing the right places. The task level bears out the ordering but not the fraction: in Table 5(a) the budget alone is worth 7.9 points (the coin's 78.9 over KVCOMM's 71.0) and the selector a further 5.4 (84.3 over that coin), so the curve bounds the selector's value only loosely. Whether the curve's ordering by unit survives at the task level is the question of Table 4(c), which §4.3 reads.

4.3 WHAT THE REPAIR RECOVERS

DENSE prefills every segment and is the ceiling in the sense that matters: no budget $\rho < 1$ can improve on ρ=1. KVCOMM is the method under study. CacheBlend (Yao et al., 2025), reproduced from its released rule inside this harness so that only its selection rule differs from BCR's, is the published selective-recomputation competitor, and a coin at the same budget is the control. The gate is γ=0.3 on the Llama benchmarks and γ=1.0 on HumanEval; every repair takes ρ=0.20, with τ=0.25 for **BCR-grow**. Within each comparison the selection condition is therefore the only variable, with one exception: GSM8K's CacheBlend pairings run at γ=1.0 against the other conditions' 0.3, a mismatch that App. D shows runs against CacheBlend.

The GSM8K recovery. In the pinned configuration, Table 5(a) has the repair recovering $84.3 - 71.0 = 13.3$ of the $84.7 - 71.0 = 13.7$ points reuse costs, level with DENSE. Against KVCOMM its discordant split is $9/49$ in that table's orientation (9 only KVCOMM solves, 49 only the repair; $p < 10^{-3}$). The run's reuse rate is 0.629, at which Equation (3), drawn through Table 7's n=150 endpoints, predicts $85.3 - 16.7 \times 0.629 = 74.8$; the repair sits 9.5 points above the line that allocation could only slide along, because its budget is spent below the call rather than among calls. The direction repeats in Table 3's reference grid: **BCR-chunk** closes the gap in every GSM8K cell and is level with DENSE at $26/25$ pooled, as App. E shows. App. H.1 withdraws a pre-registered pooled test; the n=300 run over distinct items carries this claim.

The random-selection control. At the same budget we run a coin—the control the selective-recomputation literature competes without. One run scored 82.3%, and we first read its two-point gap to the draft-scored repair as a null. Five seeds withdraw that reading. Across seeds accuracy spans 74.3 to 82.7% over the five runs App. H lists. The draft-scored repair is above all of them, three of the five paired tests separate the two, and Table 5 reports the mean. On GSM8K at 64-token chunks the selector is worth $84.3 - 78.9 = 5.4$ points, more than §4.2's curve predicted.

The layout ordering. Every repair layout is tested against KVCOMM in every cell of Table 13's grid. Of the 36 tests exactly one is significant—**BCR-single** on GSM8K at two agents, $14/4$, $p = 0.031$—and it does not survive App. H.1's correction for the comparisons it sits among. What the grid does give is direction, in the pooled splits of Table 4(c)—descriptive, because the four cells re-run the same problems. On GSM8K single rows lead KVCOMM and grown spans do not move; on MMLU the ordering inverts and weakens. HumanEval has no cost to recover: summing Table 4(d)'s $8/6$, $7/6$, $5/5$ and $8/9$ puts DENSE at $28/26$ against KVCOMM, and no repair damages it.

The published selector. Table 3 pairs CacheBlend at matched cost and at matched reuse, with Table 1(b) counting the rows. Neither pairing separates from **BCR-chunk** in any cell of Table 14 (p from 0.110 to 1.000), and Table 5(b) pools them. Ranking individual *tokens* by a layer-1 probe buys nothing here over 64-token chunks ranked by a two-token draft, though §4.2's curve predicts a gap. Table 3 also has BCR above DENSE in sixteen of its thirty-six cells, and none of them shows repair beating exact computation: the splits are level, and tokenization makes a repaired call a *different* input, free to land on either side of DENSE, as App. E explains.

The MMLU tokenization term. MMLU is graded by letter matching, and four stricter extractors put its 43-problem cost at 28–31, so its accuracies are upper bounds. In the pinned configuration, Table 5(a) shows MMLU recovering nothing significant: 3.0 points (54.7 against 51.7) on 33 discordant pairs ($12{+}21$), $p = 0.16$, while both reuse ($p < 10^{-3}$) and repair sit far below DENSE. The repair ran as designed—1,253 calls repairing 95,425 rows, five skipped—and did not help. The reuse path concatenates per-segment IDs, and that sequence need not equal the chat template's tokenization of the same text. A token-matched control keeps the sequence but removes the approximation, by forcing every call to a dense prefill of the reuse path's merged IDs. It scores 59.3%, which splits the gap in two: 6.7 points from tokenization ($66.0 - 59.3$) and 7.6 from approximation ($59.3 - 51.7$; $25/48$, $p = 0.010$). **BCR-chunk** recovered 3.0 of the 7.6 and cannot touch the 6.7, because those points are not in the cache but in which tokens the cache is *of*. On GSM8K the entry tokenizations coincide, the term vanishes, and repair recovers nearly the whole gap—the benchmark asymmetry in one line. Any reuse system that concatenates per-segment IDs pays this tax.

What the repair buys. At a budget of a fifth of the cache, BCR returns GSM8K to DENSE while 63% of calls are still served from cache, and by Table 1(c) reaches the first token $2.85\times$ sooner than DENSE does. In Table 3 a BCR layout holds the best mean short of DENSE in thirteen of fifteen columns and in every *avg* column, and the best layout's average is at or above DENSE's on all three benchmarks. CacheBlend leads no average at either pairing, and its reuse-matched pairing recomputes more than twice **BCR-chunk**'s rows per call. The draft is what pays for this: at the same budget it beats a coin by 5.4 points, and no layout damages the benchmark that has nothing to recover. Where it recovers little, on MMLU, almost half of the gap is one no budget could reach, because it is not in the cache.

5 DISCUSSION AND CONCLUSION

Cross-context KV-cache reuse is reported to be lossless. On two of three benchmarks we find otherwise. With roles pinned, reuse costs 13.7 points on GSM8K and 14.3 on MMLU (9.3–10.3 after re-grading), while HumanEval shows no measurable loss. Nor can it be avoided by choosing which calls to reuse: the value of informed selection falls from 49.5% to 10.6% to 0 as the unit of decision grows. It can be repaired: a fixed budget of exact recomputation lets BCR restore GSM8K to DENSE accuracy while keeping most of the reuse (reuse rate 0.629), and two of the three layouts reach that level; on every benchmark of the reference grid the best layout leads every reuse baseline's mean. On MMLU, however, only part of the gap is in the cache at all.

The key finding is that *where* the decision is made matters as much as *whether* it is. Our evidence spans three benchmarks and two backbones, but a single topology and a single chunk size. BCR also trades latency for accuracy: its $2.85\times$ speedup in Table 1(c) is below reuse's $7.63\times$, a trade App. J weighs. Testing the grid's ordering cell by cell, carrying the protocol to other reuse systems, and closing that latency gap are the work the same appendix lays out.

AI USE STATEMENT

In this work, we used generative AI tools to draft the structure of the paper, to revise its sentences, and to correct typographical errors. We have reviewed all AI-assisted text and take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work evaluates inference-time systems on public benchmarks (GSM8K, MMLU, HumanEval) with openly released models (Llama-3.1-8B-Instruct, Qwen2.5-Coder-7B-Instruct), used under their licenses; it involves no human subjects and collects no personal data. The reference implementation's MMLU pipeline inserts a live, uncached Wikipedia fetch into each prompt. No number we report depends on what it returned: the reference grid runs with that call disabled (App. G.2), and the pinned runs never draw the role it is gated on (App. G.1). Where we find that a published system's accuracy does not reproduce, we report it from that system's released code and state which parts do reproduce, so the comparison can be checked rather than taken on trust.

REPRODUCIBILITY STATEMENT

§4 gives the benchmarks, models, decoding (greedy, 512 new tokens) and hardware (one A100 80GB), and the two configurations every result is labelled with. App. G documents the reference configuration (App. G.1), the retrieval-disabled MMLU re-measurement—the only regime in which that benchmark reproduces (App. G.2)—and the gate sweep (App. G.3). The allocation policies and the repair are specified in App. A and App. B, and the latency harness in App. C. Every accuracy comparison is paired over the same problems and reported as its discordant split with an exact McNemar test, so each of those statistics can be recomputed from two per-problem outcome vectors. The other quantities say which test they carry: the cache-error curve is paired over six prompts and tested with a paired t, latency is a mean of paired per-call ratios, and the budget sweeps report the spread across seeds. We will release the core implementation—the repair and the allocation policies—on GitHub; the link will be given in the camera-ready version.

REFERENCES

Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Yucheng Li, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Junjie Hu, and Wen Xiao. PyramidKV: Dynamic KV cache compression based on pyramidal information funneling. In *Conference on Language Modeling*, 2025. arXiv:2406.02069.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

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

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. In *International Conference on Machine Learning*, 2024. arXiv:2305.14325.

Yuan Feng, Junlin Lv, Yukun Cao, Xike Xie, and S. Kevin Zhou. Ada-KV: Optimizing KV cache eviction by adaptive budget allocation for efficient LLM inference. In *Advances in Neural Information Processing Systems*, 2025. arXiv:2407.11550.

In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference. In *Proceedings of Machine Learning and Systems*, 2024.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In *International Conference on Learning Representations*, 2021.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Jang-Hyun Kim, Jinuk Kim, Sangwoo Kwon, Jae W. Lee, Sangdoo Yun, and Hyun Oh Song. KVzip: Query-agnostic KV cache compression with context reconstruction. In *Advances in Neural Information Processing Systems*, 2025. arXiv:2505.23416.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of SOSP*, 2023.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, 2023.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. SnapKV: LLM knows what you are looking for before generation. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

Xin Teng, Canyu Zhang, Shaoyi Zheng, Danyang Zhuo, Tianyi Zhou, and Shenji Wan. InfoFlow KV: Information-flow-aware KV recomputation for long context. In *International Conference on Machine Learning*, 2026. arXiv:2603.05353.

Shihao Wang, Jiahao Chen, Yanqi Pan, Hao Huang, Yichen Hao, Xiangyu Zou, Wen Xia, Wentao Zhang, Chongyang Qiu, and Pengfei Wang. ProphetKV: User-query-driven selective recomputation for efficient KV cache reuse in retrieval-augmented generation. *arXiv preprint arXiv:2602.02579*, 2026.

Yixuan Wang, Shiyu Ji, Yijun Liu, Yuzhuang Xu, Yang Xu, Qingfu Zhu, and Wanxiang Che. Lookahead Q-Cache: Achieving more consistent KV cache eviction via pseudo query. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, 2025. arXiv:2505.20334.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *International Conference on Learning Representations*, 2024.

Bin Yang, Qiuyu Leng, Jun Zeng, and Zhenhua Wu. CacheClip: Accelerating RAG with effective KV cache reuse. *arXiv preprint arXiv:2510.10129*, 2025.

Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. CacheBlend: Fast large language model serving for RAG with cached knowledge fusion. In *Proceedings of the European Conference on Computer Systems*, 2025.

Hancheng Ye, Zhengqi Gao, Mingyuan Ma, Qinsi Wang, Yuzhe Fu, Ming-Yu Chung, Yueqian Lin, Zhijian Liu, Jianyi Zhang, Danyang Zhuo, and Yiran Chen. KVCOMM: Online cross-context KV-cache communication for efficient LLM-based multi-agent systems. In *Advances in Neural Information Processing Systems*, 2025. arXiv:2510.12872.

Guibin Zhang, Yanwei Yue, Zhixun Li, Sukwon Yun, Guancheng Wan, Kun Wang, Dawei Cheng, Jeffrey Xu Yu, and Tianlong Chen. Cut the crap: An economical communication pipeline for LLM-based multi-agent systems. In *International Conference on Learning Representations*, 2025. arXiv:2410.02506.

Table 6: **Accuracy and reuse rate by block.** DENSE against the shipped default (γ=0.3) in blocks of 60 GSM8K problems, with that block's reuse rate. The shaded first block—what a 60-problem evaluation would see—is the only one where reuse does not lose, and the one with the lowest reuse rate.

Problems	reuse rate	DENSE	KVCOMM γ=0.3	Δ
1–60	0.537	80.0	81.7	+1.7
61–120	0.625	88.3	80.0	−8.3
121–180	0.679	90.0	85.0	−5.0
181–240	0.675	83.3	80.0	−3.3
241–300	0.675	86.7	80.0	−6.7
1–300	0.638	85.7	81.3	−4.3

Table 7: **Allocation policies on one line.** Five agents, 150 matched GSM8K problems, roles pinned; rows sorted by reuse rate. *pred.* is what Equation (3) predicts from the reuse rate alone, *resid.* the points earned beyond it. p is an exact McNemar test against DENSE; * marks $p < 0.05$. Both columns use the unrounded scores, so redoing them from the printed ones can shift the last digit.

Policy	decides from	reuse	acc. (%)	95% CI	p vs DENSE	pred.	resid.
DENSE	no reuse	0.000	85.3	[79, 90]	–	85.3	+0.0
Protect-the-Root	structural	0.456	78.7	[71, 84]	0.013*	77.7	+0.9
Random	information-free	0.454	79.3	[72, 85]	0.064	77.8	+1.6
Protect-the-Leaf	structural	0.504	77.3	[70, 83]	0.023*	76.9	+0.4
KVCOMM γ=0.3	entropy gate	0.600	75.3	[68, 82]	0.006*	75.3	+0.0

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, et al. H2O: Heavy-hitter oracle for efficient generative inference of large language models. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. GPTSwarm: Language agents as optimizable graphs. In *International Conference on Machine Learning*, 2024.

Adam Zweiger, Xinghong Fu, Han Guo, and Yoon Kim. Fast KV compaction via attention matching. In *International Conference on Machine Learning*, 2026. arXiv:2602.16284.

A ALLOCATION POLICIES IN DETAIL

The policies we test were not chosen to be exhaustive but to be *different in kind*, so that a common outcome cannot be blamed on a common blind spot. They are organised by the unit they allocate over.

A.1 ALLOCATING WHOLE AGENT CALLS

P1: the entropy gate (what KVCOMM ships). The baseline policy is Equation (2), swept from the shipped γ=0.3 to 1.0.

P2: OPENGATE, a gate on the answer's opening. Eviction's fix for a miscalibrated prompt-side score is to defer the decision past the first generated tokens. OPENGATE transplants that fix: serve optimistically, generate k tokens on the assembled cache, decide from the confidence of those k distributions, and pay a dense prefill when the draft is not trusted.

The literal transplant does not carry over, and it is worth saying why. In eviction the exact cache exists for free at the end of prefill, so deferring costs only k decode steps; in reuse the exact cache is *never computed*, and obtaining it costs the prefill that reuse exists to avoid. What can be moved is the *decision*, not the approximation.

Table 8: **Selection value by unit of choice.** One 2048-token segment, six prompts, varying only how rows are grouped and picked. Error is the relative L_2 distance of the attention output from a dense cache (0.1054 unrestored); *gap* is what informed selection removes beyond chance; the last column gives prompts where informed won and a paired t over the six. The budget keeps $\max(1, \mathrm{round}(0.2n_{\mathrm{chunks}}))$ chunks, so the last three rows are not budget-matched.

unit of choice	budget	attention	random	gap	gap / no-repair	prompts (p)
1 (single row)	20.0%	0.0375	0.0896	+0.0521	49.5%	6/6 (0.01)
16	20.3%	0.0650	0.0851	+0.0201	19.0%	6/6 (0.02)
64	18.8%	0.0763	0.0875	+0.0112	10.6%	5/6 (0.08)
256	25.0%	0.0780	0.0880	+0.0100	9.5%	4/6 (0.28)
1024	50.0%	0.0691	0.0631	−0.0060	−5.7%	1/6 (0.69)
2048 (whole segment)	100.0%	*both restore everything*		0	0%	–

P3 and P4: structural allocation. A third kind decides from the shape of the request rather than from a signal, before it starts. *Protect-the-Root* forces a dense prefill at the entry node; *Protect-the-Leaf* does the same at the deepest node. They force the identical number of dense calls and differ only in *which* node is protected, which makes them a matched pair.

P5: random allocation. Give up a fixed fraction of reuse opportunities by biased coin flip, with the probability chosen so the achieved reuse rate lands on top of the structural policies.

A.2 Allocating below the call: BCR-chunk

Chunks rather than tokens. The replacement policies above are all-or-nothing per agent call, at or above the unit of the per-segment gate they replace, and §4.2's curve points at a finer one. Per-row selection is where the choice is worth most, and it is what the selective-recomputation literature for RAG does. It is also what KVCOMM's data structures cannot express. The offset of Equation (1) is a position-indexed elementwise tensor, anchors are filtered by a length comparison, and re-rotation applies one scalar positional shift to a segment; all three assume the reused unit is a dense, contiguous, length-typed span. A contiguous chunk satisfies every one of those assumptions, and recomputing one is an ordinary prefill rather than a custom attention path.

The policy. On every call KVCOMM serves from an assembled cache, BCR inserts three steps before generation. A prompt-side score would be free at that moment—KVCOMM's own gate is one such score—but any such score ranks cached rows by the geometry of the cache rather than by what the answer turns out to need. We therefore spend k decode steps on a draft instead.

1. **Draft.** Generate k=2 tokens on the assembled cache and keep the last k query rows. The draft is then discarded: every layer is truncated back to its pre-draft length and the token counter restored. Repair and real generation therefore both begin from the cache the call was served with. The k drafted tokens are decoded a second time, now against the repaired cache, and need not come back the same. OPENGATE handles its draft the other way: it keeps the draft and continues from the returned cache (App. I). That is the case in which "no extra forward pass" holds; for BCR it does not. The served cache stops one token short of the prompt ($T = |x| - 1$), so at k=2 the kept rows are the queries of the prompt's final token and of the first generated one: one prompt-side, one answer-side. We fix k=2 and never sweep it, so §4.3's estimate bounds what one answer-side query row is worth, not what a longer draft would be.
2. **Score.** The rule of §3, with grouped-query keys expanded to query heads. Scoring a chunk by the mean of its rows' scores keeps a short trailing chunk from being penalised for its length, and unlike an eviction mask the selection cannot differ per head.
3. **Repair.** Recompute the highest-scoring ρ of the chunks exactly against the cache, and resume.

The skip rule, the recompute path, the budget's denominator and the correctness check are stated once in App. B, with Algorithm 1.

The random control. Repairing chunks and *choosing* them are separable, and §4.2 predicts the choice is worth much less at the chunk unit than at the row unit. We therefore run the same policy with the chunks picked by coin at an identical budget. Whatever separates the two rows is the draft's

Algorithm 1 BCR (Budgeted Cache Repair): repair a fixed fraction of an assembled cache, then decode.

Require: prompt $x_{1:M+1}$; assembled cache $\widetilde{K}^\ell, \widetilde{V}^\ell$ over the M cached rows (Equation (1); the reuse path slices the cache one token short of the prompt); chunk size B=64; budget ρ=0.2

if $M < 2B$ **or** $\rho M < B$ **then** serve unrepaired
partition $1{:}M$ into B-token chunks; $\mathcal{R} \leftarrow$ rows of the $\mathrm{round}(\rho M/B)$ highest-scoring ones ▷ Algorithm 2; better than a coin (Table 5)
$h \leftarrow \mathrm{embed}(x_{\mathcal{R}})$
for layer $\ell = 1, \ldots, L$ **do**
  $q, k, v \leftarrow$ layer-ℓ projections of h, rotary-encoded at $\mathcal{R}$'s true positions
  $\widetilde{K}^\ell[\mathcal{R}], \widetilde{V}^\ell[\mathcal{R}] \leftarrow k, v$ ▷ overwrite *before* attending
  $h \leftarrow$ block ℓ: causal attention of q over $\widetilde{K}^\ell, \widetilde{V}^\ell$ up to $\max P$, then MLP
end for

Ensure: rows in $\mathcal{R}$ exact, the rest assembled; decoding unchanged

Algorithm 2 Draft scoring: rank cached positions by what the answer's opening attends to.

Require: assembled cache $\widetilde{K}^\ell, \widetilde{V}^\ell$ of length M; prompt ids; draft length k=2

$\Lambda \leftarrow$ per-layer cache lengths; $c \leftarrow$ cache token counter
generate k tokens greedily on the cache, recording each layer's query projections
truncate $\widetilde{K}^\ell, \widetilde{V}^\ell$ to Λ; restore c ▷ the draft is discarded, not continued
$Q^\ell \leftarrow$ the last k recorded query rows at layer ℓ ▷ the prompt's final token, then the first $k-1$ generated ones
score positions and chunks by the draft rule of §3 ▷ grouped-query keys expanded to query heads

Ensure: chunk i scored by $\mathrm{mean}_{j \in i}\, a_j$

contribution; whatever they share belongs to repair itself. The coin condition also skips the draft entirely, so it spends k fewer decode steps per repaired call: a null between the rows would have been a dominance rather than a tie. A single coin run had suggested such a null; the five-seed control of §4.3 replaces it, so the draft is not free to remove, and App. C prices what the timed rows omit.

B SELECTIVE RECOMPUTATION

The recompute path. Repairing a chunk exactly requires everything before it, so recomputing it in isolation is wrong and recomputing the prefix again saves nothing. We run only the selected positions forward while they attend to the cache already present. Each layer does four things. The selected rows' hidden states go through the block's projections and rotary embedding. Their new keys and values overwrite their own slots *before* attention is taken, so a repaired row attending to another repaired row sees the corrected value. Their queries then attend over the whole cache under a causal mask built from absolute positions. The block output feeds the next layer. Layers proceed in order, so later layers read the corrections earlier ones made. The cost is $O(\text{selected} \times \text{total})$ attention rather than $O(\text{total}^2)$.

Correctness. Selecting every position must reproduce DENSE. Compared tensor by tensor, the two agree up to numerical precision: layer 0 is exact, and the relative difference grows with depth to a few percent at 512 tokens, because this path accumulates attention in float32 while the model's fused kernel does not. The functional check is the one that decides, and it passes—greedy generation from the two caches is token-identical over 24 tokens. We report the tensor-level discrepancy rather than tune it away.

The seam, and what the budget counts. We install the policy by wrapping `model.generate`, the same seam OPENGATE uses, and for the same reason: every state mutation in KVCOMM's reuse path happens after generate returns. We rank chunks over the whole cached prefix rather than over the reused segment alone, because segment boundaries are not visible at that seam. Budget spent on a chunk that was already exact therefore has no effect, and the row counts we report include it.

Table 9: **The gate sweep at n=60.** The threshold γ over its meaningful range on 60 matched GSM8K problems; γ=1.0 disables the entropy test, so the last row is unconditional reuse. *all*/*late* are the fraction of calls served without a dense prefill, over all 60 problems and the last 20; *calls* counts reused versus dense calls out of 240. p is an exact McNemar against the γ=0.3 default; brackets are Wilson 95% intervals.

		Accuracy			Reuse			Latency	
Method	γ	acc. (%)	95% CI	p	all	late	calls	TTFT (ms)	E2E (s)
DENSE	–	80.0	[68, 88]	–	0.000	0.000	0/240	101.5	28.33
KVCOMM	0.3	81.7	[70, 89]	–	0.537	0.662	129/111	79.2	32.00
KVCOMM	0.5	81.7	[70, 89]	1.00	0.671	0.850	161/79	82.7	36.78
KVCOMM	0.7	78.3	[66, 87]	0.73	0.787	0.875	189/51	79.6	36.70
KVCOMM	1.0	78.3	[66, 87]	0.62	0.838	0.950	201/39	78.8	36.33

Table 10: **Latency against segment length.** Five `CopyMachine` agents on synthetic prompts. The harness runs each call twice in one process—assembled cache and full prefill—so every ratio is paired, and the speedup rows are means of those per-call ratios rather than ratios of the means above them; the two differ, 7.63× against 542.7/72.9. ρ is the repaired fraction, 0 plain reuse and 1 a full prefill; the parenthetical names the repair kernel: PyTorch's scaled dot-product attention, or FlashAttention-2. *Caveat:* 12 samples per length, so the anchor pool never saturates—the best case for reuse, not its steady state. Shading marks the ρ=0.20 rows, the setting used elsewhere.

			Shared segment (symbols)			
Metric	Method	ρ	64	512	2048	4096
TTFT (ms)	DENSE	1.00	159.2	198.1	341.2	542.7
	KVCOMM	0.00	60.3	56.2	64.7	72.9
	BCR-chunk (SDPA)	0.20	102.4	112.1	155.2	229.8
	BCR-chunk (FA2)	**0.20**	**101.2**	**109.8**	**140.5**	**190.1**
	BCR-chunk (FA2)	0.10	100.8	99.0	111.1	137.0
	BCR-chunk (FA2)	0.05	89.0	100.3	108.1	119.2
	BCR-chunk (FA2)	0.30	113.8	124.6	173.4	255.1
	BCR-chunk (FA2)	0.40	126.1	144.8	212.8	304.3
Speedup vs. DENSE	KVCOMM	0.00	2.48×	3.26×	5.25×	7.63×
	BCR-chunk (SDPA)	0.20	1.56×	1.77×	2.20×	2.35×
	BCR-chunk (FA2)	**0.20**	**1.58**×	**1.81**×	**2.43**×	**2.85**×
	BCR-chunk (FA2)	0.10	1.59×	2.00×	3.08×	3.96×
	BCR-chunk (FA2)	0.05	1.80×	1.97×	3.16×	4.54×
	BCR-chunk (FA2)	0.30	1.41×	1.59×	1.97×	2.13×
	BCR-chunk (FA2)	0.40	1.27×	1.37×	1.61×	1.78×
End-to-end (s)	DENSE	1.00	14.95	14.11	14.74	15.06
	KVCOMM	0.00	14.86	13.96	14.47	14.57
	BCR-chunk (SDPA)	0.20	14.41	14.42	13.95	14.54
	BCR-chunk (FA2)	**0.20**	**14.10**	**14.22**	**14.26**	**14.64**
	BCR-chunk (FA2)	0.10	15.03	14.22	14.31	14.48
	BCR-chunk (FA2)	0.05	13.52	15.46	14.96	14.97
	BCR-chunk (FA2)	0.30	14.22	14.48	14.24	14.38
	BCR-chunk (FA2)	0.40	14.23	14.23	14.42	14.11

C LATENCY, AND WHAT THE REPAIR COSTS

Latencies come from n=60 runs with one process on an idle GPU (Table 9); the larger runs were executed sequentially, and we report no headline latencies from them—the one per-problem timing taken from an n=150 run is reported in App. H, where it does not pass a paired test.

Latency on the workload. Against DENSE's 101.5 ms, every reuse setting lands between 78.8 and 82.7 ms—a 1.2–1.3× improvement, flat in γ because the calls converted at high γ are the ambiguous ones whose cache assembly costs nearly what it replaces. The prefill being replaced is small: GSM8K questions average 59 tokens against roughly 1,000 generated across four agent calls, so prefill is

Table 11: **The repair budget swept on GSM8K, at four coin seeds.** Two agents, 150 matched problems, 64-token chunks drawn by coin so that ρ is the only variable; KVCOMM scores 77.3%, DENSE 82.7%. *Each budget is run four times*: the coin's stream advances per repaired call, so one run is one realisation of the policy, not the policy.

	Accuracy (%), by coin seed					
ρ	0	1	2	3	mean	95% CI
0.05	78.0	74.7	78.0	75.3	76.5	±2.8
0.10	76.7	76.0	82.0	75.3	77.5	±4.9
0.20	70.0	78.0	80.7	81.3	77.5	±8.3
0.30	81.3	78.0	75.3	82.7	79.3	±5.3
0.40	78.7	77.3	79.3	80.0	78.8	±1.8
KVCOMM (ρ=0)			77.3		77.3	–
DENSE (ρ=1)			82.7		82.7	–

Table 12: **The repair's layout, swept with its budget held fixed.** GSM8K, n=150, ρ=0.20, draft-scored, two and four agents; the foot rows reproduce Table 3's GSM8K columns. Only the cap on one contiguous span moves. Cells not run are dashed; b/c is against the 64-token cap, whose answers are identical to the 16-token one on all 150 problems.

		Accuracy (%)		Rows/call		b/c vs. cap 64	
Span cap	Mean len.	N=2	N=4	N=2	N=4	N=2	N=4
1	1.00	84.0	86.0	185	216	11/6 (0.33)	11/4 (0.12)
2	1.78	–	83.3	–	223	–	9/6 (0.61)
4	2.87	–	84.7	–	215	–	7/2 (0.18)
8	3.68	80.0	–	186	–	3/4 (1.00)	–
16	3.82	80.7	–	186	–	0/0 (1.00)	–
64 (**BCR-grow**)	3.83	80.7	81.3	186	221	*ref.*	*ref.*
BCR-chunk	–	82.0	85.3	172	211	10/8 (0.81)	10/4 (0.18)
CacheBlend, cost-matched	–	85.3	83.3	177	206	10/3 (0.09)	10/7 (0.63)
CacheBlend, reuse-matched	–	82.0	84.7	311	474	10/8 (0.81)	10/5 (0.30)
DENSE	–	82.7	84.7	–	–	10/7 (0.63)	9/4 (0.27)
KVCOMM	–	77.3	80.0	–	–	8/13 (0.38)	12/14 (0.85)

Table 13: **Paired statistics for Table 3.** Same runs; the discordant split and exact p at each cell. Each BCR row is split against KVCOMM, the DENSE row against KVCOMM. In every row b counts problems KVCOMM misses and the other condition solves, c the reverse—the opposite orientation to Table 5. Condition against condition is in Table 14. Shading marks the BCR rows.

		# Agents			
Dataset	Split	2	3	4	5
MMLU	DENSE vs. KVCOMM	16/12 (0.572)	15/12 (0.701)	18/14 (0.597)	18/14 (0.597)
	BCR-chunk vs. KVCOMM	4/0 (0.125)	4/3 (1.000)	5/5 (1.000)	8/4 (0.388)
	BCR-grow vs. KVCOMM	17/9 (0.169)	8/9 (1.000)	15/10 (0.424)	14/9 (0.405)
	BCR-single vs. KVCOMM	11/12 (1.000)	11/13 (0.839)	13/11 (0.839)	15/14 (1.000)
GSM8K	DENSE vs. KVCOMM	17/9 (0.169)	11/5 (0.210)	15/8 (0.210)	9/8 (1.000)
	BCR-chunk vs. KVCOMM	13/6 (0.167)	11/5 (0.210)	14/6 (0.115)	9/7 (0.804)
	BCR-grow vs. KVCOMM	13/8 (0.383)	9/9 (1.000)	14/12 (0.845)	5/10 (0.302)
	BCR-single vs. KVCOMM	**14/4** (0.031)	7/5 (0.774)	14/5 (0.064)	9/8 (1.000)
HumanEval	DENSE vs. KVCOMM	8/6 (0.791)	7/6 (1.000)	5/5 (1.000)	8/9 (1.000)
	BCR-chunk vs. KVCOMM	0/0 (1.000)	1/0 (1.000)	1/1 (1.000)	2/2 (1.000)
	BCR-grow vs. KVCOMM	7/2 (0.180)	7/3 (0.344)	7/3 (0.344)	8/4 (0.388)
	BCR-single vs. KVCOMM	7/3 (0.344)	7/3 (0.344)	7/5 (0.774)	6/4 (0.754)

≈ 0.4% of a request and the 22 ms recovered is a fifth of it. End-to-end, latency rises rather than falls, and rises *with* reuse—28.33 s DENSE against 32.00 s at the default gate, rising to 36.78 s—which

puts the per-call cost of anchor matching and offset application near 0.9–2.1 s against the 22 ms of prefill it removes.

Scaling with segment length. Table 10 repeats the measurement on synthetic prompts of controlled length. KVCOMM's time-to-first-token is nearly flat in the segment—56 to 73 ms across a 64× range—while a dense prefill of the same prompt grows from 159 to 543 ms, so the ratio grows with the segment to 7.6× at 4096 symbols, reproducing this method's headline figure. The advantage is structural: assembling a cache from anchors does not scale with the segment the way prefilling it does.

The cost of repair. The BCR rows repeat that sweep with **BCR-chunk** installed in its minimal form—random selection, no draft, the chunk attention fused; the FA2 rows route each chunk through FlashAttention-2's bottom-right-aligned causal kernel, which needs neither mask nor query folding. The premium over plain reuse is the price of the repaired rows and is linear in ρ, as the arithmetic requires: at 4096 symbols the repair costs 46, 64, 117, 182 and 231 ms at $\rho = 0.05$ through 0.40, for speedups of 4.5× down to 1.8×.

The accuracy–latency frontier. Reuse reaches the first token 7.6× sooner and pays Table 3's accuracy loss, while BCR reaches it 2.9× sooner at ρ=0.20 and, by the same table, pays none of it on GSM8K or HumanEval; on MMLU that offline grid separates no condition from DENSE either way (Table 13 for reuse, App. E for the repair), while the pinned run of §4.3 puts both far below it. For the coin-selected repair, Table 11 shows the budget buying latency but no accuracy.

What the sweep does not show. Two caveats. The end-to-end columns move by 0.6–3.3% for plain reuse and by under 6% for any repair row, buried under a generation identical in every condition, so they measure no end-to-end effect in either direction. And the sweep runs 12 samples per length, so the anchor pool never saturates—while the per-call overhead that makes reuse slower end-to-end on GSM8K grows with the number of anchors blended. Table 10 is therefore the best case for reuse, not its steady state.

Placement at a fixed budget. Everything so far varies *how much* cache is recomputed. Table 12 varies only *how it is arranged*: the budget stays $\mathrm{round}(\rho M)$ and the cap on one contiguous repaired span moves from a single token up to 64. The result is a step rather than a slope. Repairing individual rows scores above DENSE in both cells; allowing spans to grow at all costs most of that immediately, and growing them further changes nothing—the 8-, 16- and 64-token caps land within one problem of each other, and their realised mean span lengths sit within 0.2 tokens because τ, not the cap, is what stops growth. Pooled over all four agent counts (the two- and four-agent cells are those of Table 12; the other two are not tabulated), **BCR-single** leads the 64-token cap 47/27; the naive McNemar p is 0.027, but the four cells re-run the same problems (§4) and the item-level correction returns $p = 0.13$.

The table also separates the unit from the rule that picks it. CacheBlend at the matched cost is a single-row layout too, ranking rows by their layer-1 value deviation instead of by our draft score, and it lands with the 1-token cap—one cell each way, neither significant. Two unrelated selectors put the same budget on single rows and land together, which is §4.2's conclusion reached in task space.

Two qualifications. First, the comparison the body's claims rest on is against **BCR-chunk**, and there the two are indistinguishable (25/26 pooled), exactly as both are indistinguishable from DENSE: **BCR-single** is an equivalent way to spend the budget on this workload rather than a better one, and its advantage over **BCR-grow** does not carry to MMLU. Second, the ordering across the sweep is the evidence rather than any single contrast (§4.3)—read with Table 11's caution about single-run orderings, though the two settings whose caps do not bind returned identical answers on all 150 problems, which bounds this scale's run-to-run noise at zero.

Implementations. The accuracy tables in the body were produced with the earlier draft-scored, layer-serial repair; the timing rows use the coin-selected, fused one. The two are functionally equivalent—repairing every chunk reproduces DENSE's greedy generation token-for-token under both—but not equal in cost: the draft-scored repair spends k=2 decode steps per repaired call before any repair begins, and Table 10 does not charge for them. Since the control of §4.3 shows the draft

Table 14: **Each condition against BCR-chunk.** Discordant split of an exact McNemar test between the named condition and **BCR-chunk** (64-token chunks), per agent count; b counts problems the named condition gets right and **BCR-chunk** does not. Table 13 pairs against KVCOMM instead. MMLU's CacheBlend rows were re-run at the matched gate offline; GSM8K's still carry the mismatch of App. D.

Dataset	Condition	# Agents 2	3	4	5
MMLU	CacheBlend, reuse-matched	8/16 (0.152)	7/14 (0.189)	13/18 (0.473)	11/21 (0.110)
	CacheBlend, cost-matched	12/15 (0.701)	10/14 (0.541)	12/17 (0.458)	13/15 (0.851)
	BCR-grow	14/10 (0.541)	8/10 (0.815)	17/12 (0.458)	12/11 (1.000)
	BCR-single	9/14 (0.405)	10/13 (0.678)	13/11 (0.839)	12/15 (0.701)
GSM8K	CacheBlend, reuse-matched	10/10 (1.000)	5/4 (1.000)	6/7 (1.000)	4/10 (0.180)
	CacheBlend, cost-matched	10/5 (0.302)	4/8 (0.388)	4/7 (0.549)	5/10 (0.302)
	BCR-grow	8/10 (0.815)	6/12 (0.238)	4/10 (0.180)	5/12 (0.143)
	BCR-single	9/6 (0.607)	3/7 (0.344)	6/5 (1.000)	7/8 (1.000)
HumanEval	CacheBlend, reuse-matched	8/4 (0.388)	7/8 (1.000)	7/8 (1.000)	7/6 (1.000)
	CacheBlend, cost-matched	5/7 (0.774)	5/8 (0.581)	5/5 (1.000)	6/6 (1.000)
	BCR-grow	7/2 (0.180)	7/4 (0.549)	7/3 (0.344)	7/3 (0.344)
	BCR-single	7/3 (0.344)	7/4 (0.549)	8/6 (0.791)	7/5 (0.774)

Table 15: **Rows recomputed per agent call.** What each repair actually recomputes, which Table 3's reuse column cannot show, since a repaired call still counts as reused. The three layouts spend the same nominal ρ=0.20, the fixed-chunk condition quantising to whole 64-row chunks. CacheBlend's cost-matched pairing uses the same ρ; its reuse-matched pairing spends whatever that cell's reuse rate requires. MMLU is measured offline. Shading marks the BCR rows.

Dataset	Method	# Agents 2	3	4	5
MMLU	CacheBlend, reuse-matched	249	342	417	521
	CacheBlend, cost-matched	90	124	158	193
	BCR-chunk	96	123	159	191
	BCR-grow	91	136	174	209
	BCR-single	90	132	167	202
GSM8K	CacheBlend, reuse-matched	311	386	474	631
	CacheBlend, cost-matched	177	191	206	251
	BCR-chunk	172	191	211	255
	BCR-grow	186	203	221	261
	BCR-single	185	201	216	254
HumanEval	CacheBlend, reuse-matched	54	75	97	119
	CacheBlend, cost-matched	99	116	132	147
	BCR-chunk	100	121	138	156
	BCR-grow	110	133	152	172
	BCR-single	111	133	154	173

contributes to accuracy, the timed rows are to be read as the cost of the coin condition; the draft-scored configuration adds the k decode steps on top of them.

D CACHEBLEND AT MATCHED BUDGETS

Why a reproduction. CacheBlend is the published selector closest to BCR, and the comparison the body draws on it (§4.3, Table 3) is only meaningful under matched conditions: the same multi-agent workload, the same backbone, the same assembled cache served by KVCOMM, and a recomputation budget set to match BCR's row for row. CacheBlend's released implementation is a vLLM fork built for single-request RAG serving; it cannot consume the cache KVCOMM assembles, and running it as its own server would change the workload, the cache and the budget at once, leaving no way to

attribute a difference to the selection rule. We therefore reproduce its rule inside the same pipeline every other row runs in, so that the two selectors differ in nothing but how they rank the rows they recompute. The gate and budget matching below, and the ported rule's one trap (App. H), are what that reproduction required.

Setting the budget. Table 3 reproduces CacheBlend following the released implementation: the layer-1 *value* deviation ranks the tokens and the budget recomputes the top of that ranking exactly. Its third rule—recompute the request's un-cached suffix in full, outside the budget—has no counterpart here (App. H gives the reason, and what applying it anyway costs), so the budget covers the whole cache. We set it per cell so that the share left untouched equals BCR's: $\rho_{\mathrm{CB}} = 1 - r$ for **BCR-chunk**'s reuse rate r in that cell. On the Llama benchmarks that is 1.8–2.8× the cost-matched rows; on HumanEval, where reuse already runs at 84–90%, it is fewer rows (Table 15).

The gate mismatch. Both pairings were first measured at γ=1.0 against the other Llama-benchmark rows' 0.3 (on HumanEval every row runs 1.0, so the gates match there), meaning CacheBlend approximated about 90% of segments against our 63%—a gap on the axis §4.1 establishes as costly—and the cost matching held per engaged call but not in aggregate. The gap runs against CacheBlend, so on GSM8K, where CacheBlend draws, it drew at the higher gate; those rows are as first measured, and Table 3's reuse entry for them is the matched $1-\rho_{\mathrm{CB}}$, not that gate rate. On MMLU, where it trails, deficit and gate were confounded, so Table 3 carries both MMLU pairings re-run at the matched gate with retrieval disabled—which is in any case the only regime in which this benchmark reproduces (App. G.2).

At the matched gate. At the matched gate the two pairings do not separate: cost-matched scores 72.5, 71.2, 69.3 and 73.9% against reuse-matched's 69.3, 69.3, 69.3 and 68.6, with no cell significant (13/8, 13/10, 13/13, 16/8; p from 0.152 to 1.000). The direction is consistent, and it is the one that matters: the reuse-matched pairing recomputes 2.6–2.8× the rows and scores at or below the cost-matched one in every cell. Three of its four budgets—0.545, 0.516 and 0.514—land on 106 exactly, at three different agent counts, as though past about half the cache it stops mattering what is recomputed or how much. That is the budget sweeps' finding seen from the other end (Table 11), and it is why we read the cost-matched pairing as the informative one.

Against KVCOMM. The two also compare against KVCOMM. BCR leads it on GSM8K (47/24, the pre-registered pooled test withdrawn in App. H.1), while cost-matched CacheBlend leads it there by 16 problems of 600—an accuracy difference read off Table 3's cells, that pairing not being among the tabulated splits. The GSM8K figure also carries the gate mismatch above, and KVCOMM itself is the method the higher gate makes CacheBlend resemble. On the offline MMLU grid, where the gates match, cost-matched CacheBlend trails KVCOMM (54/59 pooled) and DENSE (58/78), with no cell of either pairing significant. Of the tabulated splits, Table 14 pairs CacheBlend against **BCR-chunk** and Table 13 the repair layouts against KVCOMM. Under these matched conditions **BCR-chunk** recomputes fewer than half the rows of the reuse-matched pairing and leads both pairings' means on every benchmark where reuse costs anything (Table 3); the per-cell splits above bound how much of that lead a single cell can establish.

E CELLS ABOVE DENSE PREFILL

The cells themselves. BCR is above DENSE in the *avg* column of all three benchmarks, not only in the sixteen cells §4.3 counts; **BCR-chunk**'s GSM8K margin there is +0.2, less than a single problem in 150. Two things produce them, and neither is the repair exceeding DENSE.

Resolution. The first is resolution. The pooled splits behind the cells above DENSE are level—26/25 and 30/30 on GSM8K, 49/47 on MMLU—and the gaps are one to five problems out of 150 to 160. **BCR-grow** on GSM8K is the one layout whose pooled split against DENSE is not level, at 20/40, and it falls below. The grid's one significant cell (§4.3) is against KVCOMM, the comparison those splits are drawn for, not against DENSE.

Tokenization. The second is that BCR is not DENSE minus some work. The reuse path concatenates per-segment token ids, and that sequence need not equal the chat template's tokenization of the same

text, so a repaired call is a different input rather than a degraded one. §4.3 measures that term directly: on MMLU it costs 6.7 of the 14.3-point gap, and on GSM8K the entry tokenizations coincide and it vanishes. A term that can cost 6.7 points on one benchmark can land the other way by one or two on another, which is enough to put a repaired run on either side of DENSE with nothing having been improved. On HumanEval the first account carries the cells on its own: every condition's average sits inside four points and the *reuse-matched* CacheBlend pairing is above DENSE while plain KVCOMM is below—an ordering that follows from no property of any repair, ours or theirs. One such cell arose earlier in this work and traced to a misapplied rule: with one CacheBlend rule misapplied the reproduction scored above DENSE on this same benchmark, and correcting the rule returned it to level (App. H).

Could the draft exceed DENSE? Neither account needs the repair to exceed DENSE, and there is an obvious mechanism by which it could—worth stating because we can rule it out. BCR ranks cached rows by the attention queries of a k-token draft (§3). At k=2 that is one prompt-side query row and one *answer-side* one (App. A.2)—evidence of the kind §1 faults the gate for never having—so a repaired run consults something DENSE does not, and one might expect it to place its exact recomputation better than exactness-everywhere places it.

The coin control of §4.3 settles it. The draft's signal is worth something—about 5.4 points over five coin seeds—but k decides only *where* a fixed budget $\rho < 1$ of exact recomputation lands; it cannot make that subset better than recomputing everything. On MMLU the coin means 50.0% over five seeds against the draft-scored repair's 54.7, but both sit far below DENSE's 66.0 and neither is a candidate for exceeding it.

F WHAT THE REPAIR BUYS, AND WHAT NO BUDGET REACHES

Time to first token. On a 4096-symbol segment DENSE prefills in 542.7 ms, KVCOMM assembles in 72.9 and BCR at ρ=0.20 in 190.1—7.63× and 2.85× against DENSE (Table 1(c)). The premium BCR pays over plain reuse is the price of the recomputed rows and is linear in the budget: 46, 64, 117, 182 and 231 ms at $\rho = 0.05$ through 0.40, for speedups of 4.5× down to 1.8× (App. C). The advantage is structural rather than constant— assembling a cache from anchors does not scale with the segment the way prefilling it does—so it grows with the shared segment and vanishes without one.

Throughput. We do not measure it, and on these workloads it could not separate the conditions. GSM8K's shared segment averages 59 tokens against roughly 1,000 generated across four agent calls, so the prefill reuse targets is ≈ 0.4% of a request; the end-to-end columns move by 0.6–3.3% for plain reuse and by under 6% for any repair row, buried under a generation that is identical in every condition (App. C). A throughput claim needs a workload where prefill is a larger share of the request—long shared segments, or many calls per request—and that workload is not among the three here.

The tokenization tax. The reuse path concatenates per-segment token ids, and that sequence need not equal the chat template's tokenization of the same text, so a repaired call is a *different* input rather than a degraded one. The token-matched control of §4.3 prices it: on MMLU it is 6.7 of the 14.3-point gap (66.0 − 59.3 in Table 5(a)); on GSM8K the entry tokenizations coincide and it vanishes, which is why repair recovers nearly the whole gap there. We price it on MMLU only. It is not a property of our method: any reuse system that concatenates per-segment ids pays it.

The two together. No budget reaches the tax, because it is not in the cache but in which tokens the cache is *of*. That bounds what the latency premium can buy on a benchmark where the tax is large: on MMLU at most 7.6 of the 14.3 points are inside the cache at all, and BCR recovers 3.0 of those while paying the same 2.85× it pays on GSM8K, where the tax is zero and the recovery is nearly total. A deployment choosing between KVCOMM and BCR should therefore ask first whether its entry tokenizations coincide; where they do not, the assembly path is the cheaper thing to fix.

G REPRODUCIBILITY

G.1 TABLE 3: THE REFERENCE CONFIGURATION

Table 3 is the grid the rest of the paper is read against, so its environment is given here in full. Every value below was read out of the released repository rather than chosen by us; where the repository's command-line default and its `README` disagree, we note which one is in force.

Hardware and software. One NVIDIA A100 80GB PCIe, CUDA 12.4, Python 3.10.20, PyTorch 2.4.1, Transformers 4.50.2. One process per run, no tensor or pipeline parallelism.

Backbones. `meta-llama/Llama-3.1-8B-Instruct` for MMLU and GSM8K, `Qwen/Qwen2.5-Coder-7B-Instruct` for HumanEval. Precision follows the repository's own rule—`float16` when the model name contains `llama` and `float32` otherwise—so the two Llama benchmarks run in half precision and HumanEval in single, which is what the released code does and which we leave alone. The HumanEval backbone must be passed explicitly: `run_humaneval.py` defaults `--llm_name` to the Llama model, so omitting the flag silently evaluates the wrong network.

Cache-reuse settings. The gate threshold is γ=0.3 on MMLU and GSM8K and γ=1.0 on HumanEval. Anchor-pool parameters are the dataclass defaults of `KVCommConfig`: at most 20 anchors per placeholder, eviction window 5, eight worker threads, 30 s worker timeout. These defaults are exactly the values the `README`'s reference command passes explicitly, so supplying the flags and omitting them give the same configuration; the command-line defaults for the same options are `None`, which the config object then ignores.

Topology and roles. `FullConnected` over 2 to 5 agents with a `FinalRefer` aggregation node, `decision_method FinalRefer`. Agents are `AnalyzeAgent` on MMLU, `MathSolver` on GSM8K, `CodeWriting` on HumanEval. Roles *cycle*: each agent draws a role from its class's list rather than being pinned, which is the released behaviour.

Evaluation sets. 153 MMLU questions, fixed by `limit_questions` inside the shipped runner and not adjustable from the command line (the pinned runs wrap it and set `MMLU_N` and `MMLU_ROLE`); 150 GSM8K problems; 160 HumanEval tasks. Every condition sees the same items in the same order.

Decoding. Greedy, 512 maximum new tokens, batch size 1. The generation config also carries `temperature=` 1.0 and `top_p=` 0.9, but `do_sample` is `False`, so both are inert—Transformers emits a warning to that effect on every call. Runs are therefore deterministic given a fixed input, which is what makes the retrieval channel below the only source of variation on MMLU wherever roles cycle.

The live retrieval channel. MMLU's `AnalyzeAgent` queries `en.wikipedia.org` at inference time whenever an agent draws the `Wiki Searcher` role, and splices the returned article into the prompt. The call is uncached and substitutes a placeholder when it fails. GSM8K and HumanEval never retrieve. This is the released behaviour, and the pinned configuration (Table 5) runs the code unaltered—but it pins every agent to `Knowledgeable Expert`, so no agent ever draws the role the call is gated on and the channel cannot fire there. Table 3's MMLU column is the one deviation this paper makes: the retrieval call is disabled—`search_wiki_main` replaced by a stub returning no articles—because that is the only regime in which the benchmark reproduces (App. G.2); everything else in this appendix applies to it unchanged.

Letter-match grading. MMLU is graded as `target in predicted` against a single letter, so any `A`–`D` anywhere in a free-text answer scores it correct. This too is the released behaviour, and the pinned configuration runs it unaltered.

G.2 RE-MEASURING TABLE 3'S MMLU COLUMN

Measured with retrieval on. The MMLU column Table 3 used to print was measured in the shipped configuration, which is *online*: the analysis agent splices a live, uncached `en.wikipedia.org` fetch into every prompt (App. G.1). A pre-registered re-run therefore reads a different day's Wikipedia, and it returned different numbers—the originally printed reuse costs, $12/4/9/6$ questions of 153 at two to five agents, came back as $7/3/4/11$, and BCR's margin over KVCOMM $(14/10/9/7)$ as $4/-1/-2/7$. These are two readings of an online pipeline, not two attempts at the same measurement.

Confined to MMLU. The discrepancy is confined to MMLU: the GSM8K cell at two agents returns the printed scores and split exactly $(17/9,\ p = 0.169)$, and two runs of it agree on all 150 problems individually. The environment has not drifted; the retrieval call of App. G.1 is what moves—why 13 of 459 gate decisions flipped, and why the deltas scatter in both directions.

With retrieval disabled. With the fetch disabled the pipeline is exactly reproducible: two runs of the DENSE cell agree on all 153 questions, at two agents and at five, and GSM8K, whose solver never retrieves, reproduces without any intervention at all. That offline regime is the version of this benchmark the paper reports. It returns **BCR-grow** above DENSE in three of four cells, as the original column had it and the retrieval-on re-measurement did not. Across all three measurements only the direction survives—KVCOMM below DENSE in every cell of every version.

G.3 THE γ SWEEP AT n=60

Configuration and conditions. The reference configuration of §4 on the first 60 GSM8K problems, every condition seeing them in the same order. The DENSE baseline is the `default` execution mode, which never consults the anchor pool; the KVCOMM conditions are `allow_kv_reuse` at $\gamma \in \{0.3, 0.5, 0.7, 1.0\}$, where 0.3 is the repository default and 1.0 the value its TTFT benchmark uses.

Measurement. Time-to-first-token is captured by a stopping-criteria hook around the generation call and, in the reuse path, includes cache-assembly time; model load (≈ 13 s) is excluded. Reuse rate is the fraction of the four agent calls served without a dense prefill. Runs are deterministic (App. G.1), so variation across conditions is attributable to the conditions—subject to the small-sample caveat of App. J.

Prompts differ between the two execution modes. The largest hazard, and one that silently invalidates a run: *the two execution modes do not assemble identical prompts* (§4.3). The concrete defect is a newline: on the MMLU agent the reuse path inserts an extra one before the turn delimiter in every question we probed, at the entry agent, before any approximation has occurred; on the GSM8K agent the entry prompts are token-identical. The full-repair control of that section is what exposed it, which is an argument for running that control in any such system.

The threshold flag does not reach `MathSolver`. The second hazard also silently invalidates a run. `MathSolver` drops `llm_config` on construction and falls back to the environment default, so `--kv-threshold` never arrives; the sibling agents forward it, so the defect is invisible outside the math pipeline. A first sweep run this way produced near-identical numbers at every setting—passing every superficial check while silently running at the default throughout. All results here therefore set the `THRESHOLD` environment variable in addition to the flag, and verify that each setting took effect from the rejection counter, which decreases monotonically $32, 17, 4, 0$ across $\gamma \in \{0.3, 0.5, 0.7, 1.0\}$—the arithmetic check that the threshold was applied—and by requiring an empty threshold-exceeded log at $\gamma = 1.0$.

H NEGATIVE AND NULL RESULTS

We record the directions the measurements did not support, so that they need not be re-attempted.

The coin's five seeds. §4.3 withdraws a null that one coin run supported. The five runs score 223, 231, 236, 246 and 248 of 300 matched GSM8K problems—74.3, 77.0, 78.7, 82.0 and 82.7%,

mean 236.8 or 78.9%, the value Table 5 reports. The draft-scored repair scores 253, above all five; the withdrawn single run scored 247, above four of them. A policy whose coin advances once per repaired call can land anywhere in that nine-point band, so one run bounds it from neither side.

Layout of the repair: five variants. §4.3 attributes the gap between **BCR-grow** and DENSE on GSM8K to the unit of repair—**BCR-chunk** closes it there—which makes a finer or a cleverer unit the natural next step. We tried both on HumanEval at a fixed row budget (ρ=0.20, so every variant recomputes the same 140–143 rows per call and only the layout differs), and no variant separates from a single realisation of the 64-token coin (one run per cell; Table 11 bounds how much such a cell can show): 16-token chunks by coin, 13/20; the same chunks chosen by draft attention, 16/17; variable-length spans of 1 to 64 tokens, 25/15. Within the variable-length layout, informed placement does not beat the same length distribution placed at random (24/16)—an echo of §4.2's coin result one level finer. One observation survives as suggestive rather than established: variable-length spans are nominally above DENSE in all four cells (6/3, 6/3, 8/4, 11/6; none significant, the pooled 31/16 descriptive) and hold that score at ρ=0.10, a third of the rows. A cheaper operating point is reached through the budget rather than through the layout.

Transplanting CacheBlend. CacheBlend's released implementation is a vLLM fork, so reproducing it here meant porting its rule rather than running its server, and the port has a trap worth recording. Three rules make up the method: rank tokens by the layer-1 *value* deviation; spend the budget on the top of that ranking; and recompute, in full and outside the budget, every token of the request that no cached chunk supplies. The third has no counterpart in this workload. The cache KVCOMM serves is assembled entirely from stored pieces—a per-agent prefix cache followed by rotated placeholder and prefix segments—so there is no un-cached tail, and treating the uncovered positions as one recomputes 74% of the cache: DENSE under CacheBlend's name. Applying the rule as written, the budget covers the whole cache.

The distinction matters. With the third rule misapplied, the reproduction scored *above* DENSE on HumanEval (7/1 discordant at two agents), which at first read as a property of the workload; with the rule applied correctly the same cell is within noise of DENSE at either pairing (Table 3). Any re-use of this comparison should check what the served cache is actually made of before mapping a chunk-fusion rule onto it.

Deferring the approximation. The direct analogue of deferred eviction—generate the opening on an exact cache, then swap in the approximated one—was ruled out analytically before implementation, for the reason App. A gives; partial recovery is additionally blocked by the causal entanglement of KV entries. We note it here because it is the first idea the analogy suggests, and the analysis rules it out.

HumanEval at the tighter gate. The main text reports HumanEval at γ=1.0. Repeating the four cells at the shipped γ=0.3, which lowers reuse from 84–90% to 73–82%, leaves the comparison against DENSE null in the other direction (21/32 pooled, descriptively; the naive McNemar p is 0.17, and is not a test here—the four cells re-run the same tasks). Neither gate setting produces a cost on this benchmark, so the null is not an artefact of the operating point we chose to display.

H.1 WITHDRAWN POOLED TESTS

The splits themselves. Three pooled splits are withdrawn, each for the same reason. The pre-registered 47/24 on GSM8K against KVCOMM ($p = 0.0086$, §4.3): the four cells re-run the same problems, so the pooled split is repeated measures rather than independent pairs, and the item-level correction returns $p = 0.089$. The 52/30 on GSM8K against DENSE, which is Table 13's column sums ($p = 0.020$), on the same ground; the MMLU counterpart there, 67/52, was never significant ($p = 0.199$). And 89/58 on MMLU, withdrawn twice over—it is also one of the numbers that did not survive re-measurement offline, where the same pooling returns the 67/52 above (App. G.2).

The grid's one significant cell. Of the 36 per-cell tests of a repair layout against KVCOMM in Table 13, one is significant: **BCR-single** on GSM8K at two agents, 14/4, $p = 0.031$. At $\alpha = 0.05$ the expected number of such cells under the null is $36 \times 0.05 = 1.8$, and the cell does not survive a correction for the comparisons it sits among: Bonferroni over the 36 tests gives $p = 1$, and over the

12 tests of its own benchmark and layout family, $p = 0.37$. We therefore read the grid for direction only (§4), and no claim in the body rests on this cell.

The budget sweep, read at one run per budget. Read one run per budget—on single runs that are themselves further realisations, not among the four seeds Table 11 prints—the sweep appeared to climb out of KVCOMM's deficit and flatten, 77.3% at ρ=0.05 then 79.3, 80.7, 79.3, 80.0. The seeds show that shape to lie within run-to-run variation, for the reason that table gives. This bounds what the sweep can support; it does not contradict the repair—the draft-scored condition at n=300 recovers 40 problems at $p < 10^{-4}$ (Table 5), over distinct items with surviving logs.

A latency gain not supported by a paired test. Mean time per problem falls from 34.2 s to 29.2 s under repair, which did not survive a paired test at n=150: median saving 0.27 s, faster on 83 of 150 problems, Wilcoxon $p = 0.077$. The mean is carried by a handful of slow baseline requests. We record it because a mean over a heavy-tailed paired measurement can mislead in the same way the discordant-count rule of §5 guards against on the accuracy side.

I OPENGATE IMPLEMENTATION

OPENGATE is applied as a runtime patch rather than a fork, so the measured baseline and the measured method run the same repository code. §4.1 reports it at θ=0.5, the one setting we ran there: it does not recover what reuse costs, which is the only thing the paper draws from it.

Why `generate` is the seam. The patch wraps `model.generate`. This is the correct seam because every state mutation in the reuse path—appending the response KV and token ids to shared memory, updating the anchor bookkeeping—happens *after* the generate call returns. Raising from inside generate therefore aborts a reuse attempt without leaving partial state behind, which calling the public entry point twice would not: a two-token draft issued through the normal interface registers as a two-token agent response and corrupts every downstream agent's view of the conversation.

On a call with an assembled cache. On a call that carries an assembled cache, the wrapper generates k tokens with `output_scores`, computes the mean top-1 probability, accepts the draft when it is at least θ, and either raises (rejection, caught by a wrapper on `generate_for_agent` which re-dispatches to the dense path) or continues generation from the draft's returned cache with the attention mask extended by k. Calls without a cache—dense prefills and the single-token segment probes—are passed through untouched.

The θ=0 plumbing control. The θ=0 control, reported only here, exists to test exactly this plumbing: with every draft accepted, the draft-then-continue path must be indistinguishable from single-pass generation under greedy decoding. It agrees with γ=1.0 on all 60 problems individually, not merely in aggregate accuracy.

J LIMITATIONS AND FUTURE WORK

Sample size. The matched-pair protocol of §4 proved valuable in four cases. At the smaller evaluation size, reuse appeared nearly free, the answer-side gate appeared consistently dominant, the structural policy appeared beneficial, and a 14.6% latency gain appeared measurable. Each observation was subsequently tested at n=300 or under a matched control, yielding the more robust results reported in this paper. Across these cases, the key distinction was the number of discordant pairs rather than the p-value alone: a clean b=0 among three discordant problems occurs with probability one in eight under the null. We therefore report discordant counts alongside significance tests throughout and recommend using roughly twenty or more paired observations when drawing conclusions from such comparisons.

Latency. At 4096-symbol segments, KVCOMM reaches the first token 7.6× sooner than DENSE, while BCR achieves a 2.9× speedup, with the latency premium scaling linearly with ρ and thus remaining tunable (App. C). On the three benchmarks considered here, neither speedup translates

directly into wall-clock gains: GSM8K's shared segment averages 59 tokens, so the entire prefill reuse opportunity accounts for only ≈!100,ms of a ≈!28,s request. This reflects the workload characteristics rather than a limitation of either method. The two approaches therefore offer complementary accuracy–latency trade-offs: deployments that can accommodate the accuracy loss in Table 3 can exploit KVCOMM's full speedup, while BCR retains 2.9× of the speedup and restores GSM8K accuracy to the DENSE level.

Where the evidence extends. Three benchmarks, two backbones, one topology, one chunk size, and a budget sweep on one cell establish a consistent evaluation foundation; anchor-pool capacity and the eviction window are held at the repository defaults. These settings define natural axes for extending the findings, all of which can be explored directly with the existing harness. The n=300 comparisons against DENSE are significant, and the cost effect has the same direction across every cell and measurement, providing a stable basis for studying its magnitude. The next step is therefore to characterize how this magnitude changes with the gate, sharpening the cost–accuracy trade-off rather than establishing its direction. To make these results readily interpretable and reproducible, reuse rates are consistently reported together with the gate (App. H) and averaging window (§4.1, Table 6); this reporting convention also provides a clear template for extending the evaluation to the remaining axes.

Future work. Three directions follow directly from the results. *Significance beyond one run.* The pinned GSM8K run at n=300 provides a direct test, while the reference grid orders the methods consistently across every column of Table 3. The natural next step is to validate this consistent ordering cell by cell, using pinned n=300 runs on the remaining benchmarks and backbones; because the grid cells re-run the same problems, distinct items are needed for an independent test (§4). On MMLU, the same evaluation can directly test whether a budget recovers the 7.6 points present in the cache while holding the tokenization term fixed. The pinned run already recovers 3.0 points at $p = 0.16$, providing an encouraging signal, while the offline grid does not separate the conditions sufficiently to resolve the question. A larger pinned evaluation can therefore turn this promising observation into a definitive measurement.

Generality. BCR assumes only an assembled cache and a seam at which to draft, making its protocol—matched budget, coin control, and token-matched control—directly applicable to any cross-context reuse method that concatenates per-segment ids. The same protocol also extends naturally to the topologies, chunk sizes, and anchor-pool settings held fixed here. Carrying the three quantities measured in this paper—the cost of reuse, its tokenization share, and the value of the draft—across these systems provides a direct and systematic test of how broadly the findings transfer. The existing harness already supports this extension, making broader validation a natural next step.

Latency. The gap between 2.9× and 7.6× is primarily an engineering opportunity rather than a fundamental limitation: a cheaper recompute path, an adaptive budget that shrinks as the anchor pool matures, and workloads with long shared segments or many calls per request can increase the end-to-end benefit when prefill represents a sufficiently large share of the request (App. F). Within the repair itself, two promising directions are finer-grained repair below the row level, which KVCOMM's position-indexed offset does not currently express, and a more informative random control at the single-row unit, where §4.2 shows that a selector carries roughly five times the value it carries at the chunk level. Together, these directions provide concrete opportunities to improve both the efficiency and resolution of the repair mechanism.