ReprForge: Efficient Index Rebuilding for Visual Document Retriever Upgrades

Anonymous authors
Paper under double-blind review

Abstract

Upgrading a visual document retriever usually requires re-encoding the entire collection. However, this process repeats costly visual computation, which may remain valid after the upgrade. Reusing such computation requires determining which visual results the target model needs and whether they remain valid. In this paper, we present `ReprForge`, which answers these questions by tracing how visual outputs are produced and consumed. From one forward pass, it identifies the complete visual state required by later stages and its computational dependencies, then retains this state during index construction. At upgrade time, it checks the dependencies as loaded and the processed page inputs. When both remain unchanged, the target model resumes from the retained state with newly generated text embeddings and positions; otherwise, it performs full encoding. This links reuse to the actual computation rather than a manually chosen cache boundary. On 76 released upgrade pairs, reuse decisions agree with independent bitwise checks on tested pages. On 20,946 pages under a common processing configuration and a fixed input contract, four real upgrades achieve 4.35–4.48× rebuild speedups and reduce total build and rebuild time, including the initial build and state retention, by 61%, with bitwise-identical indexes under the tested settings. These results show that exact index updates do not require repeating the full encoding process: intermediate visual computation can remain reusable even when the final document representations change.

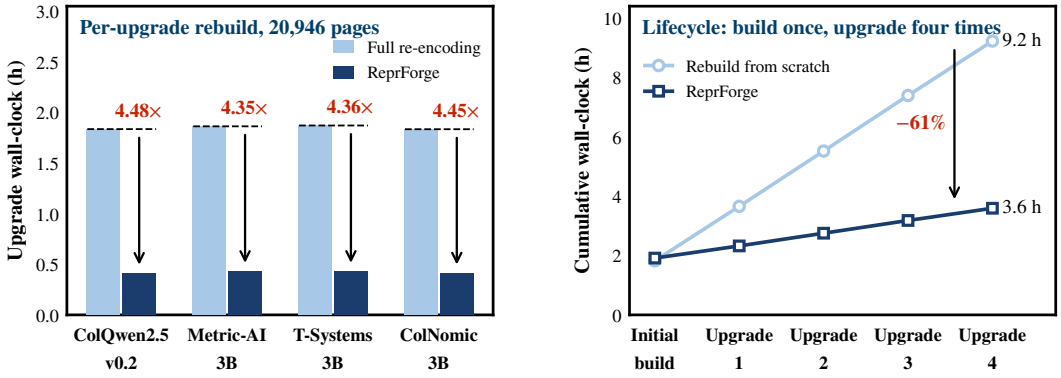

Figure 1: Index rebuilding on 20,946 real pages. (a) Wall-clock time of each of the four real upgrades, re-encoding every page from the raw image against a ReprForge rebuild from retained state; ratios in red. (b) Cumulative time of one initial build plus the same four upgrades. Both routes use a common processing configuration, with replay under a fixed input contract, and produce bitwise identical indexes.

1 Introduction

Documents contain information in text, tables, figures, and page layout. Retrieval based on extracted text can lose visual details and spatial relationships that are needed to understand a page. Visual

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.

document retrieval addresses this limitation by using vision–language models to encode page images directly (Faysse et al., 2025). In multi-vector retrievers, each page is represented by a set of vectors that capture information from different parts of the page. These vectors are computed offline and matched against query vectors through late interaction (Khattab & Zaharia, 2020). This design supports efficient search, but the stored vectors depend on the model used to produce them. When the retriever is upgraded, building an index that matches the new model usually requires encoding the entire document collection again.

Rebuilding the index repeats the full encoding process for every page, from image decoding and preprocessing to visual encoding, language-side computation, and the retrieval head. The visual branch accounts for a large share of this cost, which increases with page resolution and the number of visual tokens. As collections grow and models are updated, repeated encoding becomes a substantial maintenance cost. Existing methods reduce the storage required by multi-vector indexes through compression (Santhanam et al., 2022; Ma et al., 2025). Other approaches avoid immediate rebuilding by training new models to work with old document vectors (Shen et al., 2020) or mapping new query vectors into the old embedding space (Vejendla, 2025). However, keeping an old index usable does not produce the same representations as a full rebuild with the target model. This leaves a practical question: **Can we reproduce the target index while avoiding computation that remains unchanged across an upgrade?**

A retriever upgrade may change only part of the encoding pipeline. For example, updating a language-side adapter or the retrieval head can change the final document vectors while leaving the visual computation unchanged. Our survey of public releases shows that 107 of 151 resolvable adapter releases contain no adapter updates to the vision encoder, merger, or projections. These releases suggest that visual computation can often be reused, although changes to the base model or input processing may still make the saved results invalid. When the relevant model components and page inputs remain unchanged, the target retriever can reuse the visual results and run only the remaining stages to produce new document vectors.

Reusing visual computation requires more than saving the final output of a vision encoder. The saved state must include all visual results used by later stages, and this set differs across architectures. Some models pass only the final visual tokens to the language model. Qwen3-VL also feeds intermediate visual features into several language layers through DeepStack (Bai et al., 2025a), so the final tokens alone are not enough to resume encoding. The validity of the saved state also depends on how each page is processed. Even when the visual weights remain unchanged, different resize rules or pixel budgets can change the inputs to the visual branch. Safe reuse therefore requires identifying the complete visual state and checking both the model components and page inputs that produce it.

In this paper, we present `ReprForge`, a method that identifies, validates, and stores reusable visual computation for retriever upgrades. From one forward pass, it discovers all visual outputs consumed by later stages, including intermediate features injected into language layers, and records the parameters and buffers that produce them. This allows the system to determine what to save across architectures without manually choosing a cache location for each model. `ReprForge` records replay-state sizes, replay costs, and dependencies, exposing the storage–recomputation trade-off to the deployment. At upgrade time, it validates the saved state at two levels: the relevant parameters and buffers as loaded, and the processed input for each page. Valid state is reused with text embeddings of the target model, positions, and remaining computation; failed checks trigger full target encoding. The default path retains BF16 state for exact reconstruction. We further study PCA followed by INT8 quantization to reduce storage, measuring the resulting trade-off between state size, retrieval quality, and agreement with the target index.

Experiments show that `ReprForge` reduces rebuilding time while preserving the exact output of the target retriever. Across five retriever configurations, it identifies the required visual state in less than 1.5 seconds. On 76 released upgrade pairs, its reuse decisions agree with independent bitwise comparisons against full target encoding on the tested pages. We further evaluate four real upgrades on a collection of 20,946 pages under a common processing configuration and a fixed input contract. `ReprForge` speeds up each rebuild by 4.35-4.48$\times$, and all four rebuilt indexes are bitwise identical to those produced by full target encoding in the same execution environment. Including the initial build and the cost of saving intermediate state, total time across the four upgrades falls from 9.2 to 3.6 hours, a reduction of 61%. These results show that retaining visual state once can reduce the cost of repeated upgrades while fully preserving the target representations.

The main contributions of this paper are:

- We formulate retriever upgrades as exact reconstruction of the target index from validated intermediate results. This formulation separates the computation that must be repeated from the computation that can be reused.
- We develop a method that identifies the required visual state and its dependencies from one forward pass. It checks reuse conditions at both the model and page levels and resumes encoding with the target retriever.
- We evaluate ReprForge on five retriever configurations and 76 released upgrade pairs. Its reuse decisions agree with independent bitwise comparisons against full target encoding on the tested pages. On a collection of 20,946 pages under a common processing configuration and a fixed input contract, one initial build and four real upgrades reduce total index build and rebuild time by 61%, with bitwise-identical rebuilt indexes in the tested execution environment.

2 Related Work

Visual document retrieval and index maintenance. ColBERT introduced multi-vector retrieval, where queries and documents are encoded separately and matched through late interaction (Khattab & Zaharia, 2020). ColPali extends this design to visual document retrieval by encoding page images with a vision–language model and storing the resulting multi-vector representations (Faysse et al., 2025). Because document representations are computed offline, the stored index is tied to the model that produced it. When the retriever changes, obtaining an index that matches the target model usually requires re-encoding the document collection. Existing work reduces the storage cost of multi-vector representations through compression (Santhanam et al., 2022; Ma et al., 2025). ReprForge instead targets the rebuild itself. It retains intermediate visual state only to accelerate future upgrades; this state is not used during retrieval and can be discarded when it is no longer needed.

Retriever upgrades and representation compatibility. Existing work often avoids rebuilding an index by preserving compatibility with previously stored representations. This includes backward-compatible representation learning (Shen et al., 2020; Zhang et al., 2022; Hu et al., 2022) and methods that map new queries into an older embedding space (Vejendla, 2025). Forward-compatible training retains side information and learns a transformation from old to new embeddings (Ramanujan et al., 2022). ReprForge addresses a different objective: it reconstructs the index that the target retriever itself would produce, while reusing only intermediate computation that remains valid across the upgrade.

Reuse of intermediate computation. More generally, prior systems reuse intermediate results to avoid repeated computation. HELIX reuses materialized results across iterations of machine-learning workflows when the corresponding operators and their inputs remain equivalent (Xin et al., 2018). Materialized-view maintenance studies the more general problem of reusing stored results as their underlying computation changes (Gupta & Mumick, 1995). ReprForge considers the same reuse question inside visual retrievers, where the reused result is an intermediate visual state produced during model encoding. It identifies the visual state required by later computation and the dependencies that produce that state, then reuses it only when those dependencies and the processed page input remain unchanged. The reuse operates across modules within the retriever, with loaded dependencies checked before target-side replay.

3 Method

ReprForge reuses visual computation across two phases. When an index is first built, it identifies the visual state needed by later computation, records what produces that state, and retains the state for future upgrades. When the retriever is later updated, it checks whether the retained state is still valid for each page. A valid state is reused and the updated retriever completes the remaining computation; otherwise, the visual state is recomputed from the raw page. Figure 2 summarizes this process.

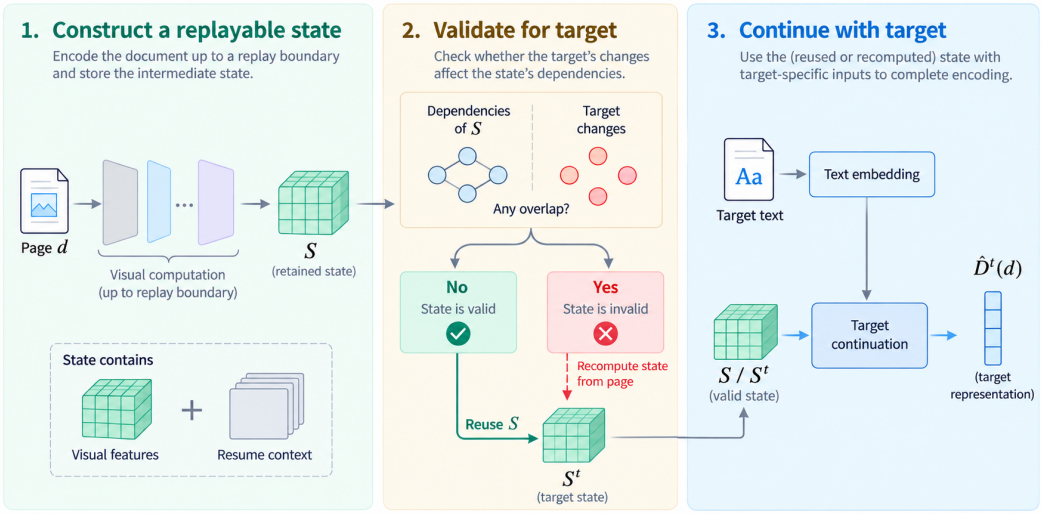

Figure 2: Overview of ReprForge. During initial indexing, the system constructs and retains a replayable visual state. At upgrade time, it checks whether the state remains valid for the updated retriever. A valid state is reused; otherwise, the visual state is recomputed from the page. The updated retriever then completes the remaining computation using its own text-side context.

3.1 Finding a replayable boundary and state

A natural way to reuse visual computation is to save the final output of the vision encoder and start the updated model from there. This works only when later computation needs no other visual information. Modern vision–language models do not always satisfy this condition. Later layers may still read visual features produced earlier in the visual branch.

ReprForge therefore does not assume a fixed cache location. It traces one forward pass and records where intermediate tensors come from and where they are later used. Tensors derived from the page image are tracked as visual. Text inputs and structural inputs, such as grids and masks, are tracked separately.

The implementation propagates these tags through the executed PyTorch operators while recording the parameters and buffers that contribute to each marked tensor. For each visual frontier tensor, it also records the producing and consuming modules and its position in the module output, so that the same interface can later be restored during replay (Appendix L.1).

For the executed path, the trace finds every visual output that later computation still reads. We call these outputs the **visual interface**. A valid replay state must contain this complete interface. Otherwise, some later stage would lose visual information that it normally receives.

Qwen3-VL shows why this matters. Its language side consumes the main merged visual tokens, but it also receives three intermediate visual features through DeepStack. These features are injected into early language layers. Saving only the final visual tokens would therefore be incomplete. Figure 3 shows the four visual outputs that must remain available.

We call the point where replay begins the **replay boundary**. At this point, the retained state replaces the visual computation that would otherwise be rerun. The retained state contains the complete visual interface together with the structural metadata needed to use it, such as the visual grid, sequence layout, and token mapping. It does not retain text embeddings or positions from the earlier retriever. The updated retriever generates those values again.

Once the replay state has been identified, the next question is whether it is still valid after an upgrade.

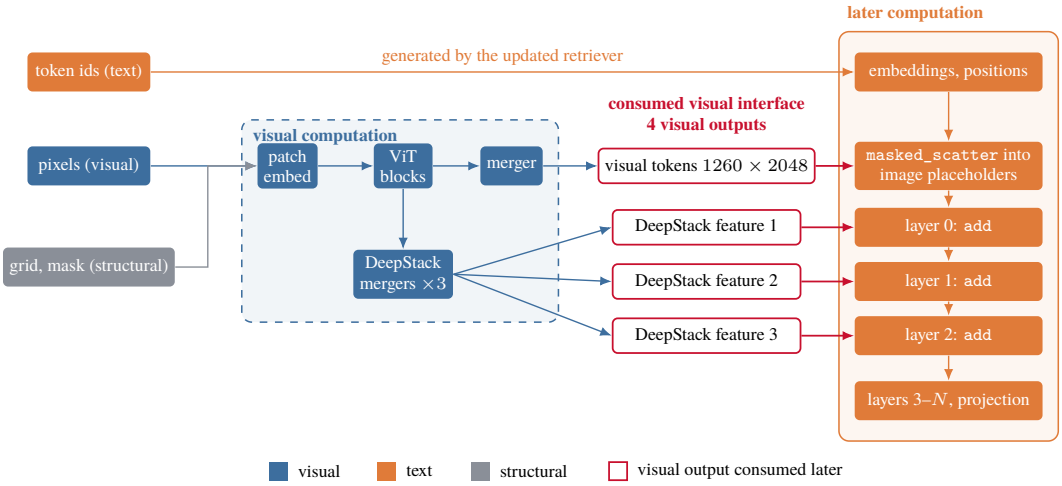

Figure 3: Visual-interface discovery on Qwen3-VL. The language side consumes not only the main merged visual tokens but also three intermediate DeepStack features. These four outputs therefore form the complete visual interface that must be retained to resume later computation.

3.2 Validating the retained state

A retained state can be reused only if the updated retriever would produce the same state for the same page. Recomputing that state only to compare it with the retained one would defeat the purpose of reuse. ReprForge therefore checks the conditions that determine the state instead.

The first condition concerns the model itself. The same trace used above also records the parameters and buffers that produce the retained visual interface. We call them the **state dependencies**. At upgrade time, ReprForge compares these values between the earlier and updated retrievers after both models have been loaded. This matters because some relevant values may be created or modified during model initialization and may not appear reliably in the serialized checkpoint.

Only changes to the state dependencies matter for this check. A change elsewhere in the retriever does not invalidate the retained visual state if it does not affect the computation that produced that state.

The second condition concerns the page input. Matching model dependencies are not enough if the page is processed differently. A different resize rule, pixel budget, or visual grid can change the input to the visual branch even when its weights are unchanged. In the general validation path, ReprForge therefore runs the updated retriever's processor and compares the actual visual input with the one recorded during initial indexing.

If the deployment keeps the raw page and processing configuration fixed across upgrades, ReprForge can use a **fixed input contract** instead of rerunning the processor for every page. In this path, the deployment environment guarantees that the visual input is unchanged.

The fixed-contract path uses the same model-dependency check and target-side replay as the general path. The only difference is how page-input equality is established. If the fixed-input guarantee does not hold, ReprForge falls back to the general validation path or to full encoding.

The state is reused only when both conditions hold: the relevant model computation is unchanged, and the input to that computation is known to be unchanged. Under these conditions and compatible execution semantics (Appendix K), the updated retriever would reproduce the same state at the replay boundary. If either condition fails, ReprForge recomputes the visual state from the raw page.

3.3 Resuming with the updated retriever

The validity conditions above apply only to the computation that produces the retained state. They do not require the entire retriever to remain unchanged. Once a state has been validated, ReprForge reuses only that visual computation and lets the updated retriever execute its own remaining stages.

At the replay boundary, the retained visual outputs replace the visual computation that would otherwise be rerun. The updated retriever then rebuilds the context that was deliberately not stored with the state, including its own text embeddings, positions, and other language-side inputs. The retained visual interface is placed where the updated model normally consumes it, including intermediate language layers such as the DeepStack injections in Qwen3-VL.

Everything after the replay boundary follows the updated retriever's normal execution path. Its language layers, adapters, normalization, retrieval projection, and retrieval head may therefore differ from those of the earlier retriever. These changes can alter the final document representation without invalidating the retained visual state because they occur after the reused computation.

This separation allows ReprForge to rebuild the updated retriever's own index rather than preserve the old one. Let $S_{\text{old}}(d)$ be the retained state for page d, and let $S_{\text{target}}(d)$ be the state that the updated retriever would produce at the same replay boundary. If

$$S_{\text{old}}(d) = S_{\text{target}}(d), \tag{1}$$

then continuing the updated retriever from the retained state gives the same result as continuing from the state it would have produced itself. ReprForge therefore reuses only validated visual computation; all version-specific computation after the boundary remains that of the updated retriever.

3.4 Retention trade-offs

A model can have more than one valid replay boundary. A later boundary can skip more computation during an upgrade, but it may also require more storage or depend on more components that future upgrades may change. ReprForge therefore separates two questions: whether a boundary is valid for replay, and whether retaining its state is worthwhile in practice.

For each candidate boundary, ReprForge records the retained-state size, replay cost, and the dependencies that determine whether the state remains reusable. These measurements expose the storage–recomputation trade-off to the deployment, which can then choose which replay states to retain according to its storage budget and expected upgrade pattern. At upgrade time, only states whose dependencies still match are considered for replay. The main evaluation retains one fixed post-vision state per page; invalid state requires full encoding from the raw image.

Our default evaluation stores retained state in BF16 and targets exact reconstruction. We additionally study PCA followed by INT8 quantization as an optional lossy storage mode. This reduces the storage footprint but no longer guarantees bitwise equality with full target encoding, so we evaluate its effect on representation agreement and retrieval quality separately.

4 Experiments

4.1 Experimental setup

We use **full target encoding** as the correctness reference. It runs the updated retriever from the raw page and produces the representation that ReprForge must reproduce. For efficiency, we compare against a **full index rebuild**, which re-encodes the entire collection after each upgrade.

Data. We use three public visual-document retrieval collections. ViDoRe v1 (Faysse et al., 2025) contributes 2,840 pages from its six splits, ViDoSeek (Wang et al., 2025) contributes 5,380 pages, and ViDoRe v3 (Loison et al., 2026) contributes 12,726 pages from five domains. Together they form the 20,946-page collection used for the deployment-scale rebuild experiment.

Metrics. At the representation level, we compare tensor shape, finite values, elementwise equality, and maximum absolute error against full target encoding. At the retrieval level, we keep the target

query representations fixed and use ColBERT MaxSim scoring (Khattab & Zaharia, 2020) with a fixed tie-breaking rule. We report top-1 agreement, ordered top-10 agreement, and target agreement at 10 (TA@10), defined as the fraction of the target's top-10 pages that also appear in the rebuilt index's top 10.

Environment and timing. Unless stated otherwise, experiments use one A100-80GB GPU, BF16 inference, batch size 1, and the same attention implementation. Bitwise equality is reported only within a fixed execution environment.

Timing scope depends on the experiment. Detailed model identifiers, frozen revisions, and timing protocols are given in Appendices B, E, and F.3.

4.2 Cross-architecture replay-state discovery

We evaluate the state-discovery procedure of Section 3.1 across different vision–language retriever architectures. The five configurations in Table 1 cover four architecture families: Qwen2.5-VL, Qwen3-VL, PaliGemma, and Idefics3/SmolVLM. Qwen3-VL is evaluated at two model sizes, and the 4B model is tested under two released wrappers.

For each configuration, ReprForge traces one forward pass and identifies the visual interface, its state dependencies, and a replay boundary. We then resume encoding from the discovered state and compare the result with full target encoding. Recovery is bitwise exact on every tested page. This provides an end-to-end check that the discovered interface contains all visual information required by later computation. Trace time is below 1.5 seconds for all configurations.

Table 1: Replay-state discovery across five retriever configurations. Interface is the number of visual outputs consumed by later computation; Dependencies is the number of parameters and buffers that produce them; Trace is the time for one traced forward pass; State is the retained size per page; Recovery reports pages whose resumed encoding is bitwise identical to full target encoding.

Architecture	Interface	Dependencies	Trace (s)	State (MiB/page)	Recovery
Qwen2.5-VL-3B	1	391	1.33	2.90	16/16
Qwen3-VL-2B	4	316	0.71	19.69	8/8
Qwen3-VL-4B (two wrappers)	4	316	0.62 / 0.25	24.61	8/8
PaliGemma-3B	1	440	0.50	4.00	8/8
Idefics3 / SmolVLM-256M	1	198	0.64	1.08	20/20

The discovered state differs across architectures. Qwen2.5-VL, PaliGemma, and Idefics3/SmolVLM each require one retained visual output. Qwen3-VL requires four because its language side also consumes three DeepStack features. This difference is reflected in the storage footprint: the Qwen3-VL states are substantially larger than the single-interface states.

The trace captures a non-persistent rotary-position buffer absent from the Qwen2.5-VL state dictionary and recovers matching Qwen3-VL-4B interfaces across released wrappers after prefix normalization (Appendix L.1).

To check the automatic discovery more directly, we also construct complete manual interfaces for Qwen2.5-VL and Qwen3-VL-4B. The manual and automatic routes use the same state capture, serialization, validation, and replay code; only the interface and dependency specification differ. In both architectures, automatic discovery matches the complete manual specification exactly: 391 dependencies for Qwen2.5-VL and 316 for Qwen3-VL-4B. Both routes also recover all eight tested pages exactly for each architecture. Thus, on these two models, the automatically discovered replay state matches the complete interface specified by an expert.

4.3 Validity across released upgrades

We apply the model-side and page-side validity checks of Section 3.2 to 76 released source–target upgrade pairs. For each pair, we independently run full target encoding on the tested pages and use bitwise equality as the reference verdict.

The decisions agree with the bitwise reference on all 76 pairs. Fifty-nine pairs are directly reusable under the processor configuration used by the audit. Another ten pairs keep the same relevant visual computation but use different processing configurations, such as different pixel budgets. These pairs become reusable when both releases are evaluated under one common processing configuration. The remaining seven pairs change parameters inside the visual computation and are therefore rejected.

Table 2: Reuse decisions across released retriever upgrades. Direct denotes pairs that pass under the processor used by the audit, including the family-base fallback when the release processor cannot be loaded. Common-config denotes pairs that pass after fixing one processing configuration for the model family. Rejected denotes pairs whose retained visual state is invalid because the relevant visual computation changes. Page counts vary by protocol; see Appendix L.2.

Source family	Targets	Direct	Common-config	Rejected	State (MiB/page)
Qwen2.5-VL-3B (colqwen2.5-v0.1)	10	4	6	0	2.90
Qwen2-VL-2B (colqwen2-v0.1)	29	25	4	0	2.17
PaliGemma mix-448 (colpali-v1.1)	9	8	0	1	4.00
PaliGemma pt-448 (colpali-v1.2)	20	20	0	0	4.00
Idefics3 256M (colSmol-256M)	6	0	0	6	1.08
Qwen3-VL 2B / 4B	2	2	0	0	19.69 / 24.61
Total	76	59	10	7	

These three outcomes separate the two validity conditions. The ten common-config pairs show that unchanged visual weights alone are not sufficient: a different processor can change the pixels or visual grid presented to the model. Conversely, the seven rejected pairs show that matching page inputs are not sufficient when the visual dependencies themselves change.

We also test the checks by deliberately violating them. Removing required DeepStack features, reusing source-side text context, or ignoring changed visual parameters all breaks exact reconstruction on the corresponding controlled examples. These controls are reported in Appendix D and Appendix C.2.

4.4 Exact target reconstruction

We compare replay directly with full target encoding at the representation and retrieval levels. On the full ViDoRe v1 collection, ReprForge reconstructs all 2,840 pages exactly: across 1,538,545,024 representation elements, the maximum absolute error is zero. The four target configurations evaluated on the 200-page gallery and the Qwen3-VL transition are also bitwise identical to full target encoding.

The retrieval results show that this is not a trivial consequence of the source and target indexes being similar. Using the same 1,033 held-out queries, the source and target indexes agree on the top-1 result for only 80.5% of queries, and no query preserves the same ordered top-10 ranking. After replay, however, all 1,033 queries recover the target's ordered top-10 exactly.

These results show that ReprForge does not preserve the source index or approximate the target representation. When the retained state passes validation, target-side replay reproduces the index that the updated retriever itself would have produced from full encoding.

Moving from the native to the common configuration reduces ColNomic 3B nDCG@10 from 0.596018 to 0.577742 on the full ViDoRe v3 HR test split, a difference of 0.018276, or about 1.83 percentage points. Common replay also achieves 0.577742: its representations and full scoring matrix are identical to common full encoding (Appendix E.1).

4.5 End-to-end lifecycle cost

We measure lifecycle cost on the 20,946-page collection from Section 4.1. We first build the source index with state retention enabled and then rebuild it for four released target retrievers. All four upgrades use the common processing configuration of 602,112 pixels and the fixed-input-contract path of Section 3.2. Under this contract, page inputs are known to be unchanged, so replay does not

repeat page decoding and preprocessing. The reported lifecycle times cover document-representation generation and publication and exclude physical ANN index construction.

Table 3: End-to-end rebuild cost on 20,946 pages. The four targets are ColQwen2.5 v0.2, Metric-AI 3B, T-Systems 3B, and ColNomic 3B. The initial build without retention is extrapolated from a measured 3,000-page run; the retained build and all upgrade measurements use the full collection. Totals are computed before rounding.

Stage	Wall-clock time (s)	Per page (ms)	Bitwise identical
Initial build	6,556	313	
Initial build with retention	6,889	328	
Four upgrades, full rebuild	26,654 (6,604–6,736 each)	315–321	
Four upgrades, ReprForge	6,043 (1,476–1,543 each)	70–73	$4 \times 20{,}946$ pages
Lifecycle (one build + four upgrades)	33,211 → 12,933		−61%

Across the four targets, the resulting speedups range from 4.35× to 4.48×.

Retention has two costs. The initial retained build takes 333 seconds longer than the estimated ordinary initial build, and it stores 63.9 GB of additional intermediate visual state for future replay. After accounting for the additional 333 seconds at the initial build, total lifecycle time falls from about 9.2 to 3.6 hours, a 61% reduction in wall-clock time. The additional storage is a separate time–space trade-off: ReprForge retains intermediate visual state so that future upgrades can avoid recomputing it.

Under the fixed input contract, replay skips both page preprocessing and the retained visual computation. Detailed per-stage timings are reported in Appendix F.3 (Figure 5).

All four rebuilt index generations remain bitwise identical to full target encoding. Per-page scaling and larger-scale extrapolations are reported in Appendix F.3.

When a fixed input contract is unavailable, rerunning the target processor for generic input validation still yields a 1.878× speedup on a separate 200-page experiment (1.818× including the one-time dependency check; Appendix E.2).

These two paths separate the benefit of reusable model computation from the cost of establishing page-input validity: the fixed contract removes per-page processing, whereas the generic path verifies the target visual input explicitly. In this separate experiment, both use the same dependency check and target-side replay.

5 Conclusion

We presented ReprForge, a method for rebuilding visual document indexes by reusing intermediate computation that remains valid across retriever upgrades. ReprForge identifies a complete replayable visual state from the executed computation, validates whether that state is still correct for the updated retriever, and lets the updated retriever complete its own remaining computation. Across multiple retriever architectures and released upgrade pairs, the discovered states support exact recovery under the tested settings, and the reuse decisions agree with independent bitwise checks. On a 20,946-page collection under a common processing configuration and a fixed input contract, four real upgrades achieve 4.35–4.48× rebuild speedups and reduce total build-and-upgrade time by 61%, including the cost of retaining state during the initial build.

The broader conclusion is that a retriever upgrade should not automatically imply recomputing the entire document encoding pipeline. A changed final representation does not mean that every intermediate result has become invalid. Instead, the useful question is which intermediate computation still satisfies the updated retriever and can therefore be carried forward safely. ReprForge makes this reuse explicit by tying it to the computation that produces and consumes the retained state, rather than to a manually chosen cache boundary. This view turns index rebuilding from an all-or-nothing operation into selective recomputation of the parts that actually changed.

Reproducibility Statement

Section 3 describes interface discovery, two-level validity, and the target-side resume. Appendices B, C.2, and H give the models, the data, the fixed execution environment, the interface adaptations, and the timing scope of each measurement; Appendix L gives the execution-trace and released-upgrade audit protocols; Appendices E and F.3 give the additional validation and lifecycle protocols; Appendix I.1 gives the public release census protocol.

All experimental results come from actual model execution, and the corresponding configurations, correctness checks, and timing records are retained. The result tables and experiment figures of this paper are compiled from those records, and the schematic figures are used to explain the execution flow. Appendix J describes how to carry out state retention, full target encoding, replay, and result comparison independently, starting from public models and data; the number of repetitions and the timing scope of each experiment are stated in the corresponding protocol.

AI Use Statement

This work used large language models to assist with discussion of the method and the experimental plan, with implementation and debugging, with analysis of results, and with drafting and revising parts of the text and figures. Automated tools were also used for experiment execution and data organization. The numerical results reported in this paper come from actual model execution, and the correctness of representation recovery was verified by independent target encoding and elementwise comparison. The authors made the decisions about the research plan and the interpretation of the evidence, and take responsibility for the content of this paper.

Ethics Statement

The experiments in this work are based on public models and existing document retrieval datasets. They involve no new human-subject research and no new manual annotation. The models, datasets, and third-party code used in the experiments remain subject to their respective licenses and terms of use.

ReprForge caches and reuses intermediate state produced while encoding documents. In a real deployment, that state may retain part of the information in the original documents, and it should therefore be placed under the same access control and data management policy as the original documents and the retrieval index.

References

Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, Chunjiang Ge, Wenbin Ge, Zhifang Guo, Qidong Huang, Jie Huang, Fei Huang, Binyuan Hui, Shutong Jiang, Zhaohai Li, Mingsheng Li, Mei Li, Kaixin Li, Zicheng Lin, Junyang Lin, Xuejing Liu, Jiawei Liu, Chenglong Liu, Yang Liu, Dayiheng Liu, Shixuan Liu, Dunjie Lu, Ruilin Luo, Chenxu Lv, Rui Men, Lingchen Meng, Xuancheng Ren, Xingzhang Ren, Sibo Song, Yuchong Sun, Jun Tang, Jianhong Tu, Jianqiang Wan, Peng Wang, Pengfei Wang, Qiuyue Wang, Yuxuan Wang, Tianbao Xie, Yiheng Xu, Haiyang Xu, Jin Xu, Zhibo Yang, Mingkun Yang, Jianxin Yang, An Yang, Bowen Yu, Fei Zhang, Hang Zhang, Xi Zhang, Bo Zheng, Humen Zhong, Jingren Zhou, Fan Zhou, Jing Zhou, Yuanzhi Zhu, and Ke Zhu. Qwen3-VL technical report, 2025a. URL https://arxiv.org/abs/2511.21631.

Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, Humen Zhong, Yuanzhi Zhu, Mingkun Yang, Zhaohai Li, Jianqiang Wan, Pengfei Wang, Wei Ding, Zheren Fu, Yiheng Xu, Jiabo Ye, Xi Zhang, Tianbao Xie, Zesen Cheng, Hang Zhang, Zhibo Yang, Haiyang Xu, and Junyang Lin. Qwen2.5-VL technical report, 2025b. URL https://arxiv.org/abs/2502.13923.

Kuicai Dong, Yujing Chang, Derrick Goh Xin Deik, Dexun Li, Ruiming Tang, and Yong Liu. MMDocIR: Benchmarking multimodal retrieval for long documents. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 30971–31005, Suzhou, China, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-main.1576.

Manuel Faysse, Hugues Sibille, Tony Wu, Bilel Omrani, Gautier Viaud, Céline Hudelot, and Pierre Colombo. ColPali: Efficient document retrieval with vision language models. In *The Thirteenth International Conference on Learning Representations*, Singapore, 2025. OpenReview.net. URL https://openreview.net/forum?id=ogjBpZ8uSi.

Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18(2):3–18, 1995. URL https://homepages.inf.ed.ac.uk/wenfei/qsx/reading/gupta95maintenance.pdf.

Weihua Hu, Rajas Bansal, Kaidi Cao, Nikhil Rao, Karthik Subbian, and Jure Leskovec. Learning backward compatible embeddings. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 3018–3028. ACM, 2022. doi: 10.1145/3534678.3539194.

Omar Khattab and Matei Zaharia. ColBERT: Efficient and effective passage search via contextualized late interaction over BERT. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 39–48, Virtual Event, China, 2020. ACM. doi: 10.1145/3397271.3401075.

António Loison, Quentin Macé, Antoine Edy, Victor Xing, Tom Balough, Gabriel Moreira, Bo Liu, Manuel Faysse, Céline Hudelot, and Gautier Viaud. ViDoRe V3: A comprehensive evaluation of retrieval augmented generation in complex real-world scenarios, 2026. URL https://arxiv.org/abs/2601.08620.

Yubo Ma, Jinsong Li, Yuhang Zang, Xiaobao Wu, Xiaoyi Dong, Pan Zhang, Yuhang Cao, Haodong Duan, Jiaqi Wang, Yixin Cao, and Aixin Sun. Towards storage-efficient visual document retrieval: An empirical study on reducing patch-level embeddings. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 19568–19580, Vienna, Austria, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.findings-acl.1003.

Vivek Ramanujan, Pavan Kumar Anasosalu Vasu, Ali Farhadi, Oncel Tuzel, and Hadi Pouransari. Forward compatible training for large-scale embedding retrieval systems. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 19364–19373, 2022. doi: 10.1109/CVPR52688.2022.01878.

Keshav Santhanam, Omar Khattab, Jon Saad-Falcon, Christopher Potts, and Matei Zaharia. ColBERTv2: Effective and efficient retrieval via lightweight late interaction. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 3715–3734. Association for Computational Linguistics, 2022. doi: 10.18653/v1/2022.naacl-main.272.

Yantao Shen, Yuanjun Xiong, Wei Xia, and Stefano Soatto. Towards backward-compatible representation learning. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 6367–6376, Seattle, WA, USA, 2020. IEEE. doi: 10.1109/CVPR42600.2020.00640.

Harshil Vejendla. Drift-Adapter: A practical approach to near zero-downtime embedding model upgrades in vector databases. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 15938–15949, Suzhou, China, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-main.805.

Qiuchen Wang, Ruixue Ding, Zehui Chen, Weiqi Wu, Shihang Wang, Pengjun Xie, and Feng Zhao. ViDoRAG: Visual document retrieval-augmented generation via dynamic iterative reasoning agents. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 9113–9134, Suzhou, China, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-main.464.

Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya Parameswaran. HELIX: Holistic optimization for accelerating iterative machine learning. *Proceedings of the VLDB Endowment*, 12(4):446–460, 2018. doi: 10.14778/3297753.3297763.

Binjie Zhang, Yixiao Ge, Yantao Shen, Yu Li, Chun Yuan, Xuyuan Xu, Yexin Wang, and Ying Shan. Hot-refresh model upgrades with regression-alleviating compatible training in image retrieval. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=HTp-6yLGGX.

A Correctness Conditions for Replay

This appendix states the conditions under which replay reproduces the target retriever's own representation. The experimental protocols are described in Appendix B, the additional validation experiments in Appendix E, and the reproduction procedure in Appendix J.

Exact reconstruction requires two properties. First, the retained state must contain every visual result that later target computation consumes. Second, that state must still be valid for the target execution. The trace establishes state completeness at source-indexing time by identifying the visual outputs consumed after the replay boundary and the parameters and buffers that produce them.

At upgrade time, model-side validity is checked by comparing those dependencies after the source and target models have been loaded. Page-input validity can be established in two ways. In the general validation path, the target processor is rerun and the actual visual input is compared with the one recorded during source indexing. Under a fixed input contract, the deployment instead guarantees that the raw page and the processing conditions that determine the visual input remain unchanged. A page is replayed only when both the model-side and input-side conditions hold; otherwise it is encoded in full from the raw image.

Replay then continues with the target retriever's own context. The target rebuilds its text embeddings, positions, masks, and other language-side inputs rather than reusing them from the source. This distinction is important because a replay implementation can still be wrong about positions, masks, loading precision, or output filtering even when the visual state itself is valid.

Every correctness reference in this paper is therefore produced by an independent full target encoding rather than derived from the replay route. Representation equality is checked before retrieval agreement: identical representations imply identical rankings under fixed queries, scoring, and tie-breaking, whereas identical rankings do not imply identical representations. Bitwise reproduction in BF16 is reported only under a fixed numerical execution environment, including device, precision, batch size, and attention implementation.

B Experimental Configuration and Evaluation Protocol

B.1 ColQwen2.5 models and data

The source is fixed to `vidore/colqwen2.5-v0.1`. The four main targets are `vidore/colqwen2.5-v0.2`, `Metric-AI/ColQwen2.5-3b-multilingual-v1.0`, `tsystems/colqwen2.5-3b-multilingual-v1.0`, and `nomic-ai/colnomic-embed-multimodal-3b`. The common base is `Qwen/Qwen2.5-VL-3B-Instruct`. Every model is loaded from a frozen snapshot, and the complete version identifiers and comparison records are retained with the reproduction artifacts. Background on the architecture is given in the Qwen2.5-VL technical report (Bai et al., 2025b).

The main controlled experiments use a common processing configuration when the goal is to isolate model-side changes. Experiments that compare native processing configurations are reported separately and do not overwrite this distinction.

Table 4: Page composition of the ViDoRe v1 collection and the 200-page multi-target subset. The full collection contains the unique images present in the corresponding ViDoRe evaluation splits rather than the complete original VQA datasets.

Collection	Multi-target subset	Full collection
ArxivQA	34	500
DocVQA	34	500
InfoVQA	33	500
TabFQuAD	33	70
TAT-DQA	33	271
Shift	33	999
Total	200	2,840

Pages are deduplicated by the SHA-256 of their image bytes and selected round-robin in a fixed collection order. The 1,033 queries used in the multi-target retrieval comparison come from a fixed 30% holdout of the first five collections, selected with the original `RandomState(0)`; Shift contributes pages but no query. The 200-page and 2,840-page galleries serve different experiments, and each experiment compares full encoding and replay on its own gallery. Because relevance coverage is partial, these subsets are used to compare routes rather than to rank target models by absolute retrieval quality.

The six ViDoRe v1 inputs are `vidore/arxivqa_test_subsampled`, `vidore/docvqa_test_subsampled`, `vidore/infovqa_test_subsampled`, `vidore/tabfquad_test_subsampled`, `vidore/tatdqa_test`, and `vidore/shiftproject_test`, as released by ViDoRe (Faysse et al., 2025). Frozen revisions and input hashes are recorded so that upstream dataset updates do not change the page composition.

The deployment-scale rebuild additionally combines 2,840 ViDoRe v1 pages, 5,380 ViDoSeek pages (Wang et al., 2025), and 12,726 pages from five ViDoRe v3 domains (Loison et al., 2026), for a total of 20,946 pages. The processing-configuration experiment uses the complete ViDoRe v3 HR test split, containing 1,110 pages and 1,908 standard queries. The generic-validation timing experiment uses 200 pages sampled at uniform intervals from the same split.

B.2 Metrics and correctness reference

Full target encoding is the correctness reference throughout the paper. It runs the updated retriever from its ordinary page input and produces the representation that replay must reproduce.

At the representation level, we record tensor shape, finiteness, elementwise equality, bitwise equality, and maximum absolute error. Selected diagnostics additionally report mean absolute error and cosine similarity.

At the retrieval level, target query representations are fixed within each comparison. Document scores use ColBERT MaxSim (Khattab & Zaharia, 2020) and a fixed corpus-ID tie-breaking rule. We report top-1 agreement, ordered top-10 agreement, and target agreement at 10 (TA@10), defined as the fraction of the target top-10 documents that also occur in the compared index's top 10. Where graded relevance is available, nDCG is reported with the gain definition specified for that experiment.

B.3 Execution environment and timing conventions

The primary ColQwen2.5 environment is Linux with Python 3.11, an A100-SXM4-80GB, CUDA 12.4, torch 2.5.0, Transformers 4.53.2, PEFT 0.15.2, and colpali-engine 0.3.12. Inference uses BF16 and batch size 1. The cuDNN backend of scaled dot-product attention is disabled and TF32 is allowed for these runs.

Timing scope is reported separately for each experiment rather than assumed to be identical across the paper. Some experiments begin from raw pages, others from preprocessed CPU tensors, and some replay measurements use warm or cold retained-state reads. Speedups are therefore compared between routes of the same experiment, not across experiments with different timing boundaries.

Model loading is excluded unless explicitly stated. One-time model-level dependency checks are measured separately from per-page timing loops. Physical approximate-nearest-neighbor index construction is not part of the reported document-encoding timings.

B.4 State storage at the high-resolution correctness setting

This measurement belongs to the high-resolution 2,840-page correctness experiment and should not be confused with the 602,112-pixel lifecycle experiment of Appendix F.3.

Storage is measured from the serialized retained-state files written during source encoding rather than inferred from token counts. The 2,840 files occupy 49,302,998,976 bytes in total, or 16.56 MiB per page on average. The smallest file is 446,072 bytes and the largest is 67,373,048 bytes.

The stored BF16 state contains visual tensors and the structural context required for replay. These numbers include serialization-container overhead but exclude the raw pages and the final retrieval

index. State size depends strongly on page resolution and the number of visual tokens, which is why storage results from different processing budgets are not directly interchangeable.

C Replay-State Discovery Across Architectures

C.1 Automatic and manual interface specification

The automatic discovery procedure is compared directly with complete manually specified replay interfaces on Qwen2.5-VL and Qwen3-VL-4B. The manual and automatic routes share the same state capture, serialization, validation, and replay implementation; only the interface and dependency specification differ.

For the ColQwen2.5 v0.1→v0.2 transition, both routes identify the same complete interface and the same 391 state dependencies. On eight tested pages, both the automatic and manual routes reconstruct the target representation exactly.

For the Qwen3-VL-4B TomoroAI→OpenSearch transition, the two routes likewise identify the same complete interface and the same 316 dependencies after wrapper prefixes are normalized. Both routes are exact on all eight tested pages.

These checks do not establish that tracing eliminates all integration work. The integrations still provide model loading, seed inputs, retrieval-head handling, and assumptions about the executed operator configuration. They instead test the narrower claim that, once a model is integrated, the trace recovers the same replay interface and dependency set as a complete expert specification on these two architectures.

The source tracing forwards take approximately 0.993 seconds for the tested Qwen2.5 configuration and 0.403 seconds for the tested Qwen3-VL-4B configuration. These measurements come from a separate run and are not used to replace the historical trace timings reported in the main architecture table.

C.2 Qwen3-VL and DeepStack

This section tests two properties that are specific to Qwen3-VL: whether the replay state remains complete when the language side consumes DeepStack features, and whether exact replay is possible when the reusable visual computation is unchanged.

The positive source is the public SauerkrautLM-ColQwen3-2b-v0.1; the target loads the public adapter rudore/colqwen3-2b on top of it. Because the adapter configuration specifies only the base-model repository, frozen snapshots of both the base and adapter are used. This is a community-published adaptation rather than an official model upgrade.

A tensor-by-tensor check covers 291 vision tensors, 6 main-merger tensors, 18 DeepStack tensors, 310 base language-model tensors, and 2 base retrieval-head tensors. All 627 base parameters are identical. The 394 loaded adapter tensors match the BF16 cast of the public artifact: 392 decoder LoRA tensors and 2 retrieval-head LoRA tensors, with rank 32, alpha 32, and dropout 0.1, evaluated with the weights unmerged. The processor, tokenizer, special tokens, prompt, resize rule, and structural-input handling agree in this combination. Source and target final representations nevertheless differ on all 10 spot-checked pages because the downstream adapter changes the retrieval computation.

The Qwen3-VL environment uses Python 3.11, torch 2.6.0+cu124, Transformers 5.9.0, colpali-engine 0.3.14, and PEFT 0.18.0. Both base and adapter use BF16 with SDPA, eval mode, random seed 0, deterministic algorithms, and TF32 disabled.

The visual model contains 24 ViT blocks. Blocks 5, 11, and 17 feed three DeepStack mergers, while the main merger produces the final visual tokens. The language side consumes both the main visual tokens and the three intermediate DeepStack features. The target rebuilds its own token embeddings, multimodal rotary positions, attention inputs, and other language-side context before continuing through its retrieval projection and normalization.

Table 5: Per-page retained tensor payload for the 100-page Qwen3-VL experiment. Every page has 1,260 visual tokens and 1,271 output tokens.

State	Shape	Dtype	Bytes/page
Main visual tokens	$[1260, 2048]$	BF16	5,160,960
DeepStack feature (each)	$[1260, 2048]$	BF16	5,160,960
DeepStack features (three)	—	BF16	15,482,880
Input IDs	$[1, 1271]$	int64	10,168
Image grid	$[1, 3]$	int64	24
Valid length	$[1]$	int64	8
Total			20,654,040

Source text embeddings, source positions, and attention-derived activations are not retained. With the tokenizer and prompt unchanged in this experiment, retained input IDs specify the sequence and image placeholders; the target recomputes text embeddings and positions. The uncompressed `torch.save` container for 10 pages occupies 206,560,148 bytes in practice, and every tensor round-trips exactly.

The Qwen3 input is the first 100 corpus IDs of a frozen ViDoRe v3 HR copy, with the 32 real queries that match this gallery. The complete replay produces the same representation as full target encoding on all 100 pages. The complete score arrays are also identical under FP32 MaxSim and stable corpus-ID tie breaking; top-1 agreement, ordered top-10 agreement, and TA@10 are all 1 on this gallery.

C.3 Incomplete-state control

To test whether the main merger output alone is sufficient, we replay the same target while retaining the main visual tokens and structural context but omitting the three DeepStack features. The target model and the rest of the execution path are unchanged.

The complete state remains bitwise identical to full target encoding. The incomplete state deviates on all three diagnostic pages, with maximum absolute errors between 0.3989 and 0.4775. This isolates state completeness: for this Qwen3-VL execution path, the three DeepStack features are part of the visual interface and cannot be replaced by the final merger output alone.

C.4 Diagnostics of the replay boundary

To test whether the fixed post-vision boundary gives up a substantial amount of reusable computation, we compare several candidate replay positions on Qwen2.5-VL.

The first diagnostic uses two pages from each of the six ViDoRe v1 collections and four public targets at the same validated base and processor. Each route has two warmups and three repetitions with rotated execution order. The raw route covers preprocessing, host-to-device transfer, model execution, and device-to-host transfer. The replay route covers the warm state read, transfer, suffix computation, and output transfer. Initial state writing, validation, and physical ANN construction are excluded.

Across the four targets, full encoding, the fixed post-vision route, and a later fused route take 274.745, 64.648, and 63.945 ms per page, respectively. The fused state saves only a further 0.703 ms per page, about 1.09% of the fixed replay time, while increasing the average serialized state from 2.911 to 2.972 MiB per page. The pre-projection state is invalid for all four targets and is not counted as a legitimate saving.

We further repeat the comparison on the 200-page subset. Each target is loaded independently, and the source records states at processor, pre-merger, post-vision, fused, and pre-projection positions. Each route is timed three times per page.

Table 6: 200-page replay-boundary diagnostic. Times are milliseconds per page. The fused boundary shows no stable material gain over the fixed post-vision boundary.

Target	Full encoding	Post-vision	Fused
ViDoRe v0.2	270.876	63.096	62.425
Metric-AI	272.305	65.728	65.090
T-Systems	268.136	65.745	65.000
ColNomic	278.757	64.373	64.586

The mean serialized sizes at processor, pre-merger, post-vision, fused, and pre-projection are 12.8090, 6.9767, 2.7984, 2.8582, and 2.8413 MiB per page. The first four positions are consistent on all tested target-page conditions for which the dependencies remain valid. The pre-projection position is rejected because the language-side computation has changed. These diagnostics motivate the fixed post-vision boundary used by the main Qwen2.5 experiments: later replay is possible in this setting, but its measured benefit is small.

D Validity Checks and Controlled Failures

D.1 Controlled interventions on Qwen2.5-VL

The 200-page mechanism experiment fixes the intervention size in advance and uses every page of the frozen sample. The release control is a real adapter transition; the remaining interventions isolate one validity or reconstruction requirement at a time.

The text-embedding intervention adds 0.125 to alternating coordinates of the non-image token rows. The vision intervention multiplies `patch_embed.proj.weight` by 1.01 in BF16. The merger intervention adds 0.125 to alternating coordinates of `visual.merger.mlp[2].bias`. The position ablation sets replay position IDs to zero.

Every condition is compared with the full target encoding of that same condition.

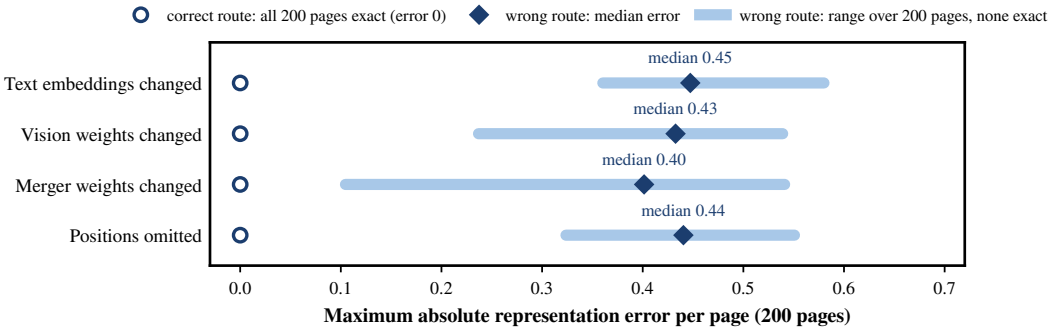

Figure 4: Controlled mechanism comparison on 200 pages. Correct replay or fallback has zero representation error. Keeping source-side context, forcing reuse after an upstream dependency change, or corrupting positions causes nonzero error on every tested page.

Table 7: Controlled validity and reconstruction conditions. Every denominator is 200 pages; all outputs have matching shapes and are finite.

Condition	Correct resume / fallback	Incorrect route
Real adapter transition	200/200	old fused text: 200/200
Target text embeddings changed	200/200	old fused text: 0/200
Vision patch weights changed	200/200	forced old state: 0/200
Merger bias changed	200/200	forced old state: 0/200
Position-resume ablation	200/200	all-zero positions: 0/200

The first row verifies that keeping the source-side fused text can happen to preserve this particular real transition, but the intervention rows show why source context is not part of the replay contract. Target-side text and position reconstruction is required in general. Likewise, changes to the vision or merger dependencies require fallback to full target encoding rather than forced reuse.

D.2 A Qwen3-VL rejection case

We also examine the community transition from `goodman2001/colqwen3-v0.1` to `goodman2001/colqwen3-v0.2`. The processor files are identical and all 628 parameter tensors agree in name, shape, and dtype, but their values differ in 252 of 291 vision-tower tensors, 5 of 6 merger tensors, 15 of 18 DeepStack tensors, 298 of 311 language-model tensors, and 2 of 2 retrieval-projection tensors.

The validator therefore rejects reuse of the retained post-vision state. A three-page forced-reuse control that deliberately ignores this decision produces a maximum absolute error of 0.5283, a mean per-page absolute difference of 0.0617, and a minimum flattened cosine of 0.4594. No speedup is reported for this rejected pair.

A larger 100-page forced-reuse control uses the target input processing and the same visual-token budget. None of the 100 pages is bitwise identical. The maximum absolute error has a median of 0.4736 and a maximum of 0.5664; ordered top-10 agreement is zero, TA@10 is 0.81875, and top-1 agreement is 0.65625.

A separate safe-fallback experiment uses the target default processor and never reads the source state after the model-level check rejects reuse. The full reference and the two fallback implementations agree, showing that rejection preserves correctness by returning to full target encoding.

E Additional Validation Experiments

E.1 Native and common processing configurations

The main lifecycle experiment uses a common 602,112-pixel processing configuration so that repeated upgrades share a fixed page-input contract. To measure the quality effect of that configuration separately from replay, we evaluate `nomic-ai/colnomic-embed-multimodal-3b` on the complete ViDoRe v3 HR test split.

The target snapshot is fixed to revision `86627b4a9b0cade577851a70afa469084f9863a4`, with Qwen base revision `66285546d2b821cf421d4f5eb2576359d3770cd3`. The native configuration uses 1,003,520 pixels and the common configuration uses 602,112 pixels. The test contains 1,110 pages and 1,908 standard queries. Retrieval quality is computed with the standard graded relevance judgments, linear graded gains, and stable corpus-ID tie breaking.

Table 8: Effect of the processing configuration on the full ViDoRe v3 HR test split. Replay is compared with full encoding under the same common target configuration.

Route	nDCG@5	nDCG@10
Native full encoding	0.571344	0.596018
Common full encoding	0.550714	0.577742
Common replay	0.550714	0.577742

Moving from the native to the common configuration reduces nDCG@10 by 0.018276, or about 1.83 percentage points, on this target and split. Common replay introduces no further difference: all 1,110 page representations and the complete score matrix are identical to common full encoding. Native and common query representations are also bitwise identical for all 1,908 queries.

This experiment separates two effects. The processing configuration itself changes document retrieval quality on this target, while replay exactly reconstructs the specified common target and introduces no additional approximation relative to it.

E.2 Cost of generic input validation

The lifecycle headline uses a fixed input contract. We separately measure the more general path in which the target processor is rerun and the actual visual input is compared with the source record.

The full route decodes and processes the page and executes the complete target model. Generic validation replay reruns the target processor, obtains the visual input that would enter the target visual branch, compares it with the source record, and resumes from retained state when the check passes. Fixed-contract replay does not rerun the processor because input invariance is supplied by the deployment contract.

The experiment uses the same ColNomic 3B target as Appendix E.1 with the 602,112-pixel common configuration, on 200 pages sampled at uniform intervals from ViDoRe v3 HR, and runs three paired rounds with rotated route order. Retained-state reads are warm, and compressed image bytes reside in RAM. Each timed route includes the relevant page processing or retained-state read, host/device transfer, target execution, and output-vector write with fsync. Model loading and the one-time target dependency check remain outside the per-page timing loop.

Table 9: Per-page validation cost on 200 ViDoRe v3 HR pages. Values are mean $\pm$ sample SD across three complete paired rounds.

Route	ms/page	Speedup
Full target encoding	316.563 ± 1.455	$1.00\times$
Generic validation replay	168.538 ± 0.842	$1.878\times$
Fixed-contract replay	72.348 ± 0.236	$4.376\times$

The one-time target dependency check takes 1.1167 seconds. Charging that check once to the 200-page run changes the effective speedups to $1.818\times$ for generic validation and $4.062\times$ for the fixed-contract path. All 1,800 timed outputs across the three routes and three rounds reproduce the corresponding full target representations exactly.

These timings are not substituted into the 20,946-page lifecycle result because the protocols differ: this experiment uses warm retained-state reads and a 200-page sample, whereas the lifecycle experiment uses its own end-to-end storage and rebuild protocol.

F Reconstruction and Lifecycle Cost

F.1 Full-collection exact reconstruction

The 2,840-page ViDoRe v1 correctness experiment compares replay with an independent full target encoding. Across 1,538,545,024 representation elements, the maximum absolute error is zero. Retrieval comparison on the 1,033 held-out queries likewise recovers the target ordered top-10 ranking exactly.

This agreement is not explained by source and target indexes being nearly identical. Before replay, source and target agree on the top-1 result for only 80.5% of these queries, and no query keeps the same ordered top-10 ranking.

F.2 Timing scope of the full-collection correctness run

The full-collection ColQwen2.5 correctness experiment is a single paired pass after warmup. Full target encoding averages 2.4349 seconds per page and replay averages 0.2932 seconds per page.

This timing scope includes image decoding and processing for the full route, or the retained-state read for replay, together with metadata reconstruction, GPU execution, and transfers. It excludes model loading, initial state retention, model-level validation, retrieval scoring, output serialization, and physical ANN construction. These timings are therefore used only within this experiment and are not combined with the lifecycle or generic-validation timing results.

F.3 Deployment-scale lifecycle experiment

The deployment-scale corpus contains 20,946 pages after 12 parquet collections are deduplicated by SHA-256 of their image bytes, removing 1,851 duplicate pages. Pages are ordered by interleaving the collections.

The experiment consists of separate processes for the source initial build with and without retention, full re-encoding of each target, replay rebuilding of each target, and bitwise verification. Four target upgrades are evaluated: ColQwen2.5 v0.2, Metric-AI 3B, T-Systems 3B, and ColNomic 3B.

All four use the common processing configuration of 602,112 pixels and the fixed-input-contract path. The corpus is frozen by image-byte hashes, and the processing conditions that determine the visual input remain fixed across the measured upgrades. The replay loop therefore checks the retained processing contract and state-file integrity but does not rerun the image processor or rehash newly generated visual tensors for every page. The generic-validation experiment in Appendix E.2, by contrast, reruns the target processor and directly compares the resulting visual-branch input.

Timing is in-process wall-clock time. The source-retention path includes model loading, page decoding and preprocessing, visual and language-side computation, retained-state serialization, file fsync, final-vector writing, generation sealing, and atomic publication.

The replay path includes model loading, cold retained-state reading with the page cache evicted beforehand, integrity checking, deserialization, validation of the fixed processing contract, target-side resume computation, vector writing, generation sealing, and publication. The model-level dependency check runs once before the rebuild begins.

The published generation contains serialized document vectors and associated metadata. Construction or maintenance of a physical approximate-nearest-neighbor search structure is outside this measurement.

Each stage performs one warmup on its first page before entering the measured page loop. The same page is then processed again inside the measured loop, so every published generation still contains all 20,946 pages. Warmup is excluded from the loop time but included in the reported in-process wall-clock time.

The retained state is stored as one file per page and occupies 63.9 GB in total, or approximately 2.91 MiB per page. This is additional intermediate visual state retained for future replay rather than the final retrieval index itself.

For every target, full encoding and replay produce separate generations that are compared page by page. Across the four targets, all $4 \times 20{,}946$ page representations and all corresponding representation elements are identical.

Table 10: Cumulative page-loop time of the end-to-end rebuild at selected collection sizes, in seconds.

Route	1,000 pages	3,000 pages	10,000 pages	20,000 pages	20,946 pages
Initial build with retention	322	984	3,256	6,553	6,874
Full re-encoding (v0.2)	312	950	3,147	6,292	6,595
ReprForge rebuild (v0.2)	68	205	698	1,391	1,459

The four full re-encodings take 6,608, 6,707, 6,736, and 6,604 seconds. The corresponding replay rebuilds take 1,476, 1,541, 1,543, and 1,483 seconds.

For each target, speedup is computed as the full-rebuild wall-clock time divided by the corresponding replay-rebuild wall-clock time. These ratios range from 4.35× to 4.48×.

The initial source build without retention is measured directly on 3,000 pages. It takes 946.08 seconds, of which 938.47 seconds belong to the page loop, or 312.82 ms per page. Scaling this measured loop rate to 20,946 pages and adding the measured 3.06-second model load and 0.82-second sealing and publication cost gives the reported 6,556-second initial-build estimate.

The retained initial build and all four target rebuilds are measured on the complete 20,946-page collection. The initial build with retention takes 6,889 seconds. State retention therefore adds 333 seconds of initial wall-clock time relative to the estimated ordinary initial build.

Across the four later upgrades, full rebuilding takes 26,654 seconds in total, whereas replay takes 6,043 seconds. Replay therefore avoids 20,611 seconds of rebuild time across these upgrades.

Including the 333-second initial retention overhead, complete lifecycle wall-clock time decreases from 33,211 seconds for one ordinary build followed by four full rebuilds to 12,933 seconds for one retained build followed by four replay rebuilds. This corresponds to approximately 9.2 hours versus 3.6 hours, a 61% reduction in measured and extrapolated lifecycle wall-clock time.

The 63.9 GB retained state is a separate storage cost and is not subtracted from wall-clock time. The lifecycle result should therefore be interpreted as a time–space trade-off: additional intermediate state is retained so that later upgrades can avoid repeating validated visual computation.

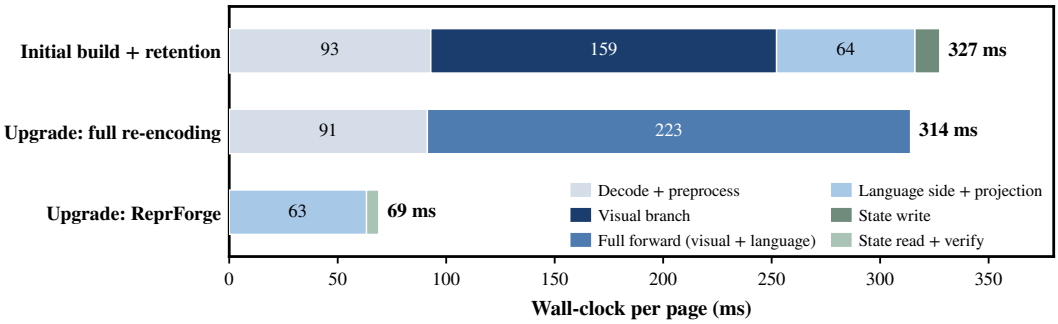

Figure 5: Per-page time by stage in the 20,946-page rebuild. Full rebuilding repeats page decoding, preprocessing, and visual computation. Under the fixed input contract, ReprForge skips these stages and instead reads the retained state before running the remaining target-side computation.

The per-page breakdown explains the measured speedup. Full re-encoding takes about 315 ms per page, including approximately 91 ms for decoding and preprocessing and 223 ms for the model forward pass. About 159 ms of that forward pass is spent in the visual branch.

Under the fixed input contract, replay skips the page preprocessing and the retained visual computation. It spends approximately 6 ms per page on the cold state read, integrity checking, and deserialization, followed by approximately 63 ms on the remaining target-side computation.

The model-level dependency check runs once per upgrade and takes approximately 1.5 seconds in this protocol. It is not repeated for every page.

Across the reported checkpoints, cumulative average per-page loop time varies by approximately 1.6% for full encoding and 2.3% for replay. Across the measured full-corpus stages, the difference between total in-process wall time and page-loop time ranges from 12.00 to 19.56 seconds, including setup and warmup. Over the measured range, the rebuild cost is therefore approximately linear in collection size.

For scale only, extrapolating the measured per-page rates gives approximately 8.7 hours for a full rebuild and 1.9 hours for replay on 100,000 pages, with about 0.31 TB of retained state. At one million pages, the corresponding extrapolations are approximately 87 and 19 hours, with about 3.1 TB of retained state. These are linear extrapolations from the measured rates rather than additional measured runs.

F.4 Retention overhead and state reads

A separate 200-page retention experiment measures the cost of capturing and writing the replay state during the source build. Both routes run the full source model and write and fsync the final page vectors; the retention route additionally captures, serializes, and fsyncs the visual state.

The two routes average 315.89 and 323.73 ms per page, respectively. The three paired passes increase source-build time by 2.77%, 2.56%, and 2.11%. The mean retained state is 2.7985 MiB per page at the 602,112-pixel budget.

A separate target pass gives 307.99 ms per page for full target encoding and 64.88 ms per page for replay, with exact agreement on all 200 pages. These values belong to this diagnostic protocol and are not substituted into the lifecycle table.

We also compare warm and local cold retained-state reads. At the 602,112-pixel budget, full encoding, warm replay, and cold replay take 276.61, 62.50, and 65.79 ms per page. At the largest tested 12.8M-pixel budget, they take 2023.60, 257.58, and 262.99 ms per page. Remote fetches and network transfer are not measured.

F.5 Illustrative storage–compute sensitivity

The paper does not claim a universal economic break-even point. Storage and compute prices vary by deployment, and retained-state size differs substantially across architectures and input resolutions.

For sensitivity analysis only, we express a break-even number of valid upgrades per year as

$$N_{\text{break}} = \frac{12bp_s}{(t_{\text{full}} - t_{\text{reuse}})p_g/3600}, \tag{2}$$

where b is retained GB per page, p_s is storage cost per GB-month, and p_g is compute cost per hour. Example prices used in the historical calculation are assumptions rather than live vendor quotations.

This analysis is intentionally illustrative. It excludes several deployment-specific terms, including request overhead, remote transfer, retention duration, and physical index maintenance. In particular, a Qwen3-VL state can be much larger than the Qwen2.5 state used by the lifecycle experiment, so a deployment should substitute its own storage and transfer model rather than treat the lifecycle speedup as a universal cost reduction.

G Supplementary Experiment on State Compression

This section uses a separate historical ColQwen2 v0.1→v1.0 protocol to study lossy state storage. It is not part of the exact-replay evaluation elsewhere in the paper.

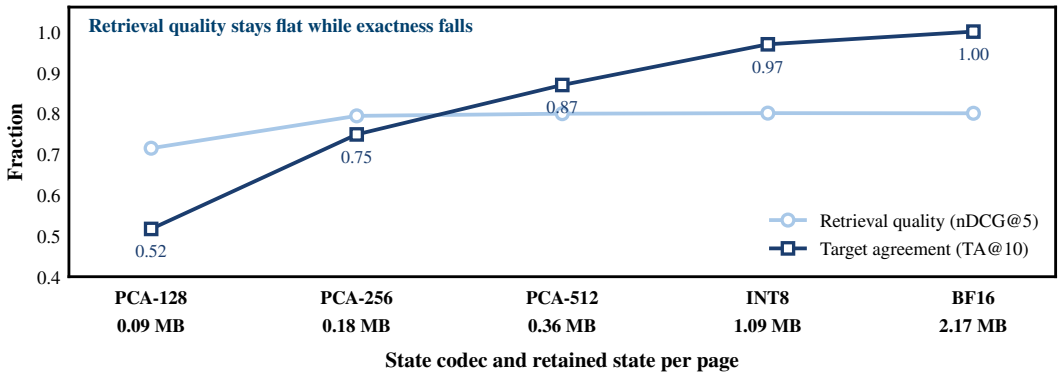

Figure 6: Historical state-compression experiment. The horizontal axis lists the storage schemes with decimal MB per page. PCA labels denote PCA followed by INT8. TA@10 measures agreement with the target index, while nDCG@5 measures retrieval quality on this gallery. Retrieval quality is not flat across all compression strengths: PCA-128 reduces nDCG@5 relative to BF16.

Table 11: Five storage schemes under the same historical compression protocol. These are descriptive results on a fixed collection.

Storage scheme	MB/page	TA@10	nDCG@5
BF16	2.173043	1.000000	0.800054
INT8	1.087936	0.968925	0.800419
PCA-512 + INT8	0.363589	0.869313	0.799069
PCA-256 + INT8	0.182502	0.748306	0.793712
PCA-128 + INT8	0.091958	0.516941	0.714529

The gallery contains pages from the six ViDoRe test collections together with additional MMDocIR distractors (Dong et al., 2025). It uses one relevant page as a proxy label rather than complete graded relevance judgments.

INT8 and PCA-512 + INT8 preserve nDCG@5 close to the BF16 value on this protocol, while TA@10 decreases as compression becomes stronger. The distinction is important: retrieval quality can remain similar even when the compressed state no longer reconstructs the target's retrieval behavior exactly. Exact-replay experiments elsewhere in the paper therefore use BF16 retained states.

H Additional Released Transitions

H.1 Qwen3-VL-4B under native configurations

The source is `TomoroAI/tomoro-colqwen3-embed-4b` and the target is `OpenSearch-AI/Ops-Colqwen3-4B`. All 315 serialized visual parameters agree in name, shape, dtype, and content hash. The native-configuration validation additionally compares every visual parameter and registered buffer as loaded, including non-persistent FP32 rotary-position buffers, together with the visual configuration and executed implementation.

Each model uses its own default processor. The experiment uses the 100 smallest corpus IDs of ViDoRe v3 HR. Although the raw processor containers have different shapes, after wrapper unpacking the pixels and grid entering the visual branch are identical on all 100 pages. The target rebuilds its own text context.

Full encoding and replay take 195.61 and 170.49 ms per page under this native-config protocol. The timing includes image decoding, default preprocessing, per-page visual-input hashing for replay, warm state reads, host/device transfer, model execution, and output transfer. Model loading, the one-time model fingerprint, final vector serialization, and physical ANN construction are excluded.

Counting the target fingerprint once, the 100-page update is 8.90% faster; counting both source and target fingerprints gives 4.65%. The result is intentionally reported separately from the fixed-input-contract lifecycle experiment because the native processors require page-level validation.

H.2 Fixed-target-processor probe

An earlier probe uses the target processor on both source and target sides, fixing the visual input before model execution. On 100 pages, all three repetitions agree exactly. Full and replay routes take 129.00 and 85.05 ms per page, about 1.52×, and the retained state occupies 24.64 MiB per page.

This probe excludes the processor and page-input validation and therefore does not replace the native-configuration result above.

H.3 Native-input audit of ColQwen2.5 releases

The ColQwen2.5 native-input audit uses the original 200 pages and each release's own frozen processing configuration. Numeric pixel budgets, tokenizers, and prompts are not overridden.

Compared with source v0.1, the actual visual input matches on 15 pages for v0.2, 200 pages for Metric-AI, 15 pages for T-Systems, and 30 pages for ColNomic. Matching page inputs alone do not establish complete model compatibility, and recovery is not forced where the input differs.

I Public Release Audit and Census

The public-release analysis serves two different purposes. Static repository inspection identifies candidate reuse opportunities, while dynamic validation determines whether a particular transition is actually safe to replay. The two should not be interpreted as the same test.

The detailed dynamic audit protocol is given in Appendix L.2.

I.1 Public release census protocol

The broader census searches repositories by model-family names. After quantized and converted formats are excluded by name, 312 repositories enter inspection and 296 have resolvable safetensors headers.

The resolvable releases comprise 151 adapters, 108 merged releases, 32 base-or-full releases, 3 partial releases, and 2 adapter-plus-full releases. Among the 151 adapters, 107 show no visual-side LoRA in the inspected tensor names and configurations, while 44 do.

Absence of visual-side LoRA is only a candidate signal. The base weights, processor, DeepStack modules, or other loaded values can still differ. Candidate relations are therefore established from available tensor hashes, declared bases, and processing fields, and then passed to dynamic validation.

The 34 candidate clusters with more than one member contain 159 releases. Twenty-seven clusters containing 137 releases expose partial processing fields; seven clusters containing 22 releases do not. These fields do not fully specify normalization, tokenizer behavior, prompts, or run-time execution semantics.

Among 18 historical version pairs examined by the static program, 5 have matching sampled weights and recorded processing fields, 2 differ in processing fields but remain candidates under a fixed configuration, 1 has unknown processing information, 7 show visual-weight differences, and 3 lack sufficient information. The historical program's label *eligible* is interpreted here only as *candidate*; the actual reuse verdict comes from the dynamic checks in Appendix L.2.

J Independent Reproduction

A reproducer should keep correctness, retrieval behavior, and efficiency as separate checks.

1. **Fix inputs and model versions.** Use the frozen model, processor, page, and query versions of the relevant experiment. Compare loaded dependency values rather than relying only on serialized checkpoint metadata.
2. **Generate and retain the source state.** Run the source execution, capture the complete visual interface and required structural context, and store the dependency record. Qwen3-VL requires both the main visual tokens and the three DeepStack features.
3. **Establish page-input validity.** For the generic validation path, rerun the target processor and compare the actual visual input with the source record. For a fixed input contract, verify that the deployment assumptions guaranteeing the same raw page and processing conditions hold.
4. **Execute the target routes independently.** The full route runs the target from the ordinary page input. The replay route loads the validated state and rebuilds target-side text, positions, masks, and other context before executing the remaining target computation. The full reference must not be generated from replay intermediates.
5. **Compare representations before retrieval.** Check shape, finiteness, equality, and numerical error page by page. Then fix target queries, scoring, and tie breaking and compare document scores and rankings.
6. **Measure timing and storage with explicit scopes.** Report which preprocessing, state reads, transfers, validation, writes, and one-time checks belong to each timing loop. Measure retained-state storage separately from final-index storage and from wall-clock savings.

The Qwen2.5 multi-target comparison, full-collection reconstruction, controlled interventions, Qwen3 recovery, generic-validation timing, and lifecycle experiments intentionally use different page sets and timing boundaries. Their results should therefore be interpreted within their own protocols rather than stitched into a single synthetic benchmark.

K Implementation Scope and Asset Use

The tracing and retain-and-resume mechanisms are architecture-agnostic once a model family is integrated. The trace takes seed inputs from an executed forward pass and returns the visual frontier, its dependency set, and candidate cut modules. Capture and resume then operate on those discovered locations.

Each model family still requires integration code that loads the released checkpoint, adapter, processor, and retrieval head and presents the page in the format expected by that implementation. The evaluated Qwen2.5-VL, Qwen3-VL, PaliGemma, and Idefics3/SmolVLM configurations are integrated in this way.

The trace describes the executed path. A single forward pass does not prove completeness for unseen data-dependent branches. Dependency and page-input equality also assume unchanged upstream operator semantics and compatible execution configuration. The current replay mechanism requires the consumed visual interface to appear as module outputs or parameter-free transformations of those outputs; architectures that fuse visual and textual computation inside an indivisible operator fall outside the evaluated scope.

Timing protocols differ across experiments. The deployment-scale lifecycle result uses one common pixel budget and a fixed input contract, while native-processing and generic-validation experiments include different amounts of page preprocessing and input validation. Remote storage, network transfer, production upgrade frequency, and physical ANN maintenance are not part of the main lifecycle measurement.

The experiments use public models, existing datasets, and third-party implementations. Code, configuration files, frozen version identifiers, and measurement records should be handled separately from redistribution of model weights or raw documents. Retained states may encode information derived from the original pages and should therefore follow the same access-control and data-management requirements as the source documents and retrieval index.

L Experiment-Specific Protocols

This appendix documents the execution tracing, released-upgrade audit, and pixel-budget sweep. Processing-configuration and validation-cost comparisons are reported in Appendix E, and the deployment-scale lifecycle protocol in Appendix F.3. Appendix A states the correctness conditions these experiments test, and Appendix J gives the reproduction checklist. Frozen model revisions, configurations, and per-page measurement records are retained with the reproduction artifacts.

L.1 Execution tracing and manual comparison

The trace is implemented with PyTorch operator dispatch and observes one executed forward pass. Seed tensors are marked according to their role: processed pixels are visual, the token sequence is textual, and the visual grid, masks, and modality identifiers are structural. Tensor marks propagate through operators, while parameters and buffers that contribute to the retained visual state are recorded as dependencies.

For each visual frontier tensor, the trace records the operator and module that produce it, the later operator and module that consume it, and its position within the module output structure. Forward hooks additionally record the execution order of the replay modules so that the same interface can later be restored during resume.

Generic retention registers hooks at the discovered replay modules and stores only the output leaves that belong to the visual interface. Generic resume replaces the corresponding visual computation with the retained tensors and reconstructs the target-side context around them. Dependencies are

compared after the models have been loaded rather than inferred only from serialized checkpoint metadata.

The two Qwen3-VL-4B publishers evaluated in this paper wrap equivalent model components under different module-name prefixes. Each side is therefore traced independently, and prefixes are normalized only when comparing modules that play the same execution role.

On Qwen2.5-VL, the dependency set includes a non-persistent rotary-position buffer that participates in the visual computation but is absent from the serialized state dictionary. The manual comparison results and model-integration scope are given in Appendix C.1.

L.2 Released-upgrade audit

The released-upgrade audit applies dynamic validation to candidates from the census of Appendix I.1.

The family audit covers 74 pairs from the census of Appendix I.1 across five model families whose visual branches share a common base: Qwen2.5-VL-3B, Qwen2-VL-2B, PaliGemma mix-448, PaliGemma pt-448, and Idefics3 256M. The source of each family is the corresponding official ViDoRe release: `vidore/colqwen2.5-v0.1`, `vidore/colqwen2-v0.1`, `vidore/colpali-v1.1`, `vidore/colpali-v1.2`, and `vidore/colSmol-256M`. Targets are successor versions resolved by the census and are loaded according to their released adapter or full-weight configuration. Frozen revisions and release metadata are retained with the experiment records.

For the family audit, 20 pages are sampled at equal intervals from the 20,946-page corpus. The sample spans nine collections and includes the 3308×2339-pixel ViDoSeek pages. The source retains state under two processing settings: its own processor and a common family configuration. The common configuration uses 602,112 pixels for the Qwen families; PaliGemma and Idefics3 use fixed resolutions, so the two settings coincide.

For each target, we load the model, trace its interface, compare the dependencies with those of the source after normalizing cut-module prefixes where necessary, and then perform page-input validation. Under the native path, each target uses its released processor. Under the controlled common path, source and target use the same processing configuration. Pages that pass both model-side and input-side validation are replayed and compared bitwise with independent full target encoding.

A family transition is classified as a *direct pass* when all tested pages validate under the processor used by the audit and replay exactly. The audit uses the target's own processor when it loads successfully and otherwise falls back to the family base processor. It is a *common-configuration pass* when native page inputs differ but replay is exact under the controlled common configuration. It is *rejected* when a dependency producing the retained state has changed.

The 10 common-configuration targets expose released pixel budgets of 602,112 for five targets, 802,816 for two, and 862,400, 1,003,520, and 12,845,056 for the remaining three. These differences explain why an unchanged visual model does not by itself imply page-input compatibility.

The released version chains illustrate both cases. For example, colqwen2 v0.1 to v1.0, colqwen2.5 v0.1 to v0.2, and colpali v1.2 to v1.3 contain reusable transitions.

Among the seven rejected family pairs, `vidore/colpali-v1.2` changes 439 of the 440 recorded dependencies relative to `colpali-v1.1` after replacing the base model; forced reuse produces a maximum absolute representation error of 0.415. Two ColSmol releases from Nayana change 36 vision-side dependencies and produce maximum errors of 0.436 and 0.453 when reuse is forced. Four fully fine-tuned ColSmol releases from vijay-ravichander change 78 dependencies beginning at the patch embedding and produce maximum errors between 0.045 and 0.056.

Table 2 additionally includes the two Qwen3-VL transitions described in Appendix C.2 and the native-configuration experiment of Appendix H, giving 76 released pairs in total. The page counts differ by protocol. Successful family transitions use the 20-page validation subset, the Qwen3 generic-resume checks use eight pages, and the seven rejected family pairs use three-page forced-reuse controls only to demonstrate what happens when a rejection is ignored. The larger Qwen3 experiments are reported separately in Appendices C.2 and H.

Forty-six releases in the census are published only as merged weights and cannot be loaded as adapters. For these releases, we inspect the safetensors headers and retrieve the visual-branch tensors required for comparison. Shape, dtype, and raw tensor bytes are compared with the corresponding family base. The Qwen2-VL family base is published in FP32 while several derived releases are BF16, so this comparison is performed after the corresponding BF16 conversion. All 46 inspected merged releases preserve the visual branch of their family base under this comparison.

Static agreement is not itself a replay verdict. It identifies a candidate relation that must still pass the loaded-model dependency check and the page-input check before retained state is reused.

L.3 Budget sweep of the encoding stage

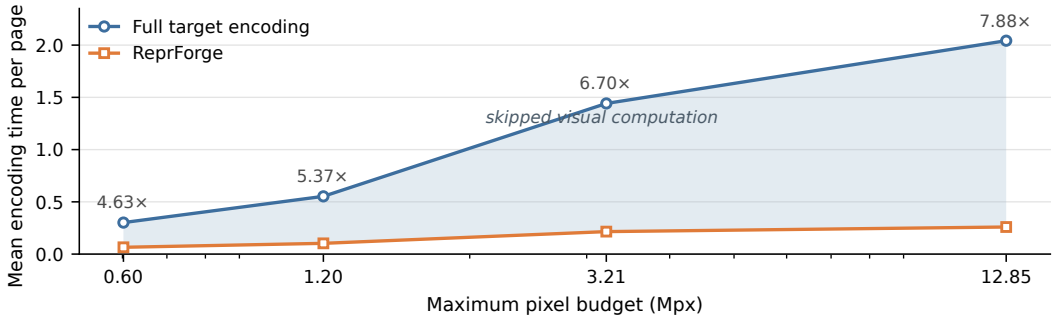

Figure 7: Page re-encoding time under controlled pixel budgets. The transition is ColQwen2.5 v0.1 to v0.2 on the same 200 pages, with three paired passes per budget. The shaded region denotes the measured encoding-time difference between the two routes. This experiment measures the document-encoding stage rather than the complete lifecycle protocol.

The sweep uses four controlled visual-input budgets between 602,112 and 12,845,056 pixels. Full encoding rises from 0.302 seconds per page at 602,112 pixels to 2.042 seconds per page at 12,845,056 pixels. Replay rises from 0.065 to 0.259 seconds per page over the same range.

The retained state also grows with visual resolution, from 2.80 MiB per page at the lowest budget to 14.64 MiB per page at the highest. Thus larger inputs increase both the amount of visual computation that can be skipped and the state that must be retained.

The encoding-stage speedup increases from 4.63× at approximately 0.60M pixels to 7.88× at 12.85M pixels. These values characterize this controlled encoding-stage experiment and should not be substituted for the end-to-end lifecycle speedups reported in Appendix F.3.