

Fine-Grained Content-Addressed Computation Reuse for Document Processing Pipelines

Master Thesis



Fine-Grained Content-Addressed Computation Reuse for Document Processing Pipelines

Master Thesis
September, 2026

By
Lingxiao Yin

Copyright: Reproduction of this publication in whole or in part must include the customary bibliographic citation, including author attribution, report title, etc.

Cover photo: Vibeke Hempler, 2012

Published by: DTU, Department of Engineering Technology and Didactics, Lautrupvang 15, 2750 Ballerup, Denmark Denmark
<https://engtech.dtu.dk/>

Approval

This thesis has been prepared over five months at the Department of Engineering Technology, Technical University of Denmark (DTU), in fulfillment of the degree of Master of Science in Computer Science and Engineering.

Lingxiao Yin - s242610

Lingxiao Yin

.....
Signature

2026.09.27

.....
Date

Abstract

Repeated submissions and document revisions can cause unchanged content to be processed multiple times by the same expensive external service. This thesis investigates to what extent fine-grained, content-addressed reuse can reduce such repeated work in an OCR-based PDF-processing pipeline. It presents a version-aware, page-level reuse mechanism in which each PDF page is rendered, canonicalized, and identified by the SHA-256 digest of its canonical representation. The cache key combines this digest with the canonicalization version and the OCR model identifier, and only exact matches are reused. Cached page results are retrieved from storage, while one representative page of each distinct cache miss is copied into a temporary PDF that is submitted in a single OCR request. Cached and newly generated page results are then mapped back to their original positions to reconstruct the document-level output.

The mechanism was implemented in a production AI-assisted review system and evaluated on an aggregate snapshot of its cache, covering approximately 38 days of a non-public subset of production traffic. In this subset, the cache contained 69,462 entries and recorded 24,035 hits among 93,497 observable distinct-key lookups, a page-level hit rate of 25.71%. The recorded hits constitute a lower bound on the avoided external OCR page operations. At the document level, 1,798 of 5,885 associated reviews (30.55%) benefited from at least one reused page, and 594 of them exhibited partial page-level reuse that a whole-file cache could not exploit. Reuse was concentrated in a small set of pages and showed strong short-term temporal locality.

These findings show that page-level content addressing can avoid a substantial share of repeated OCR work while preserving the provider’s document-level interface and the original page order. More generally, the approach can be applied wherever an expensive computation can be split into stable units that are processed independently of each other, identified by their content together with the processing configuration, and put back together in their original order.

Acknowledgements

Lingxiao Yin, MSc Computer science and engineering, DTU
Creator of this thesis template.

I would like to express my sincere appreciation to my industry supervisors, **Yong Du** and **Lele Cao**, for their valuable guidance, support, and assistance. Their help has been essential to the completion of this thesis and to my development. I am grateful for the time they dedicated to discussing my work, sharing their experience, and providing valuable feedback whenever I encountered challenges.

I want to extend my gratitude to my supervisor, **Lei You**, Assistant Professor in the Department of Engineering Technology, for his guidance and support throughout this thesis. I am also deeply grateful to my supervisor, **Birger Andersen**, Professor in the Department of Engineering Technology, for his invaluable insights and assistance during this thesis.

I am also sincerely thankful to my parents for their unwavering understanding and support throughout my studies, and to my friend Tang for encouraging me and standing by me during difficult times.

During my master's studies in Denmark, I experienced the loss of both of my grandfathers. Although I was unable to be with them during their final days, I will always carry their memory with me. Living and studying abroad has taught me to face challenges independently and has made me stronger. As this chapter of my life comes to an end, I hope that the days ahead will bring greater peace and happiness.

Statement

Generative AI tools were used to assist with code refinement and language polishing during the preparation of this thesis. All AI-generated suggestions were reviewed and revised by the author. The author takes responsibility for the final text, the implementation and analysis, and the conclusions presented in this thesis.

Contents

Preface	ii
Acknowledgements	iv
Statement	v
1 Introduction	1
2 Related Work	4
2.1 Memoization, valid reuse and versioning	4
2.2 Reuse in data-intensive and AI workflows	4
2.3 Content addressing and deduplication	5
2.4 Canonicalization and hash-based identity	6
2.5 Workload locality and cache retention	6
2.6 Document similarity and PDF processing	7
2.7 Summary	7
3 Problem Formulation and Reuse Algorithms	9
3.1 Problem Statement and Scope	9
3.2 Reuse Algorithms	13
4 Page-Level Reuse for PDF Processing	16
4.1 Baseline Full-Document OCR Workflow	16
4.2 Mechanism Design	16
5 Production Implementation	20
5.1 Implementation of the Original Workflow	20
5.2 Implementation of the Reuse Mechanism	21
5.3 Implementation Validation	28
6 Evaluation	30
6.1 Data and Measurement Method	30
6.2 Workload Reuse	31
6.3 Avoided Cost and Overhead	33
6.4 Summary	35
7 Discussion and Limitations	36
7.1 Discussion	36
7.2 Limitations	37
7.3 Future Work and Broader Applicability	38
Bibliography	39
A Appendix	43
A.1 SHA-256	43

1 Introduction

The economics of digital content are becoming increasingly asymmetric. Generative artificial intelligence (AI) has reduced the time and effort required to produce and revise text and other digital artifacts, thereby also lowering the practical cost of generating new submissions. In a controlled experiment involving professional writing tasks, access to a generative language model reduced task completion time by 40% while increasing output quality by 18% [1]. Evidence from real-world deployment points in the same direction. In a large-scale field study of customer support agents, access to generative AI assistance increased worker productivity by approximately 15%, with the largest gains among less experienced workers [2]. These findings indicate that generative AI makes the production and revision of digital content faster and more accessible. However, a lower cost of producing a digital artifact does not imply a corresponding reduction in the downstream cost of processing, analyzing, and verifying it.

Reliable processing, analysis, and verification remain resource-intensive. A submitted document may need to be validated, rendered, converted into a machine-readable representation, analyzed by one or more models, and checked against application-specific requirements before it can be used. Each of these stages consumes computational resources and processing time. Stages delegated to external services may also incur direct financial charges. Prior work has shown that large-scale NLP workloads can produce considerable financial and environmental costs, which motivates treating efficiency as a first-class system objective rather than as an implementation detail [3]. When an unchanged unit of content enters the same processing context more than once, re-executing the same expensive transformation creates cost without producing new information.

This problem is especially visible in document-processing systems. PDF remains a common representation for scientific and professional documents, yet it encodes drawing operations rather than explicit document structure. So a conversion pipeline is usually required before its content can be searched or analyzed. Research on recovering document structure from low-level PDF printing commands illustrates that robust conversion requires more than extracting embedded character strings [4]. In an AI-assisted review workflow, the uploaded PDF must be transformed into a representation that downstream models can consume. In the CSPaper system examined in this thesis, submitted PDFs are processed by Mistral OCR, whose response contains page-structured JSON results that are subsequently assembled into document-level Markdown [5]. The service is billed according to the number of pages processed.¹ Before the work described in this thesis, every submission was sent to the OCR service in full. Repeatedly invoking OCR for an unchanged page consumed time, compute capacity, and paid service usage. Even though an acceptable result for the same page might already exist.

Repeated work occurs because document identity and content identity are not equivalent. A user may upload the same file more than once. Or may submit a revised PDF in which only a title page, an appendix, or a small subset of pages has changed. A cache tied to the review identifier cannot recognize the same PDF when it is submitted again as a new review. A cache tied to a hash of the whole file recognizes exact re-uploads. But a change to a single page produces a new hash, so the results for all other pages are lost. The relevant question is not only whether a document has been seen before, but

¹<https://mistral.ai/pricing/>

which independently processable units within it have already been processed under a compatible configuration.

Reusing previously computed results is a well-established idea. Memoization returns a stored output when a function is called again with a matching input. Later work shows that whether this pays off depends on how equality is defined, which dependencies are recorded, and how much space the stored results occupy [6]. Content-addressed systems such as Venti complement this idea by deriving an object’s identifier from its content, so that identical data is recognized regardless of its name or the object that contains it [7]. Together, these ideas suggest that repeated OCR work can be avoided if pages are identified by their content rather than by the document in which they appear. Chapter 2 discusses these lines of work in detail.

How well content identification works depends on the granularity of the units being identified. Coarse-grained identification has low bookkeeping overhead but loses reuse whenever a small edit changes a large enclosing object. Very fine-grained identification may discover more redundancy, but increases hashing, indexing, storage, and lookup costs. For the examined workflow, a PDF page is a domain-aligned unit because of the following reasons: the OCR service returns page-structured results and charges per page, document revisions often affect only a subset of pages, and page results can be reassembled in their original order.

This thesis combines content-based identification with page-level granularity in a reuse mechanism for OCR in PDF processing. Each uploaded PDF is rendered page by page, and each rendering is passed through a versioned canonicalization pipeline (grayscale conversion, contrast normalization, smoothing, and intensity quantization). The SHA-256 digest [8] of the resulting canonical bytes serves as the page content identifier. The cache key additionally contains the canonicalization version and the OCR model identifier. This also allows results produced under different rendering configurations and OCR models to be distinguished in later analysis. The mechanism deliberately chooses exact matches of canonical bytes because a small visual change, such as an altered digit in a table or a flipped sign in an equation, may carry meaning in scientific documents. Reusing an earlier result in such a case would silently corrupt the review input.

If a valid entry exists, the stored OCR result is loaded. Otherwise, one representative page of each distinct missing key is copied into a temporary PDF, which is submitted in a single OCR request. The page thus becomes the unit of identification, reuse, and cost accounting, whereas the PDF remains the unit of transport. Cached and newly computed page results are then mapped back to their original positions, adapted to those positions, and concatenated into document-level Markdown in the same format as before.

Prior work does not establish how much exact page-level redundancy occurs in the operational workload examined here. It also does not quantify the gross OCR work avoided by such reuse or the overhead threshold below which the mechanism would produce a net benefit. A technically correct cache may provide little value if most pages are unique or if its overhead approaches the cost of OCR. Conversely, a moderate page-level hit rate can be valuable when each miss invokes a comparatively expensive external service.

Accordingly, this thesis investigates the following main research question:

To what extent can fine-grained, content-addressed reuse reduce repeated OCR work in a real-world PDF workload?

To answer this question, the thesis measures how much page-level reuse occurs in a

production workload and how this reuse is distributed across cached pages, reviewed documents, and time. From these measurements, it derives a lower bound on the external OCR work that was avoided. It also states the break-even condition that the reuse overhead must satisfy for the avoided work to become a net saving.

The evaluation is based on an aggregate snapshot of the retained cache state, collected over approximately 38 days from a non-public subset of production traffic. The snapshot contains 69,462 cache entries and 24,035 recorded cache hits, which corresponds to 93,497 observable distinct-key lookups and a page hit rate of 25.71%. The recorded hits constitute a lower bound of 24,035 avoided external OCR page operations. At the document level, 1,798 of 5,885 associated reviews (30.55%) benefited from at least one reused page; 594 of them exhibited partial page-level reuse that an all-or-nothing whole-file cache could not exploit. Reuse is concentrated in a small set of pages. Reused entries also tend to have short creation-to-last-modification spans, indicating short-term temporal locality in the observed workload.

The contributions of this thesis are threefold.

First, it formalizes page-level OCR result reuse by distinct content keys and persistent-cache hits and derives the corresponding break-even condition.

Second, it designs and implements a version-aware page-level reuse pipeline in an operational system. Combining deterministic rendering and canonicalization, SHA-256-based page identity, selective batch OCR of cache misses, and order-preserving result integration.

Third, it evaluates the deployed mechanism using an aggregate snapshot of production workload data. The evaluation quantifies observable page reuse, review-level coverage, reuse concentration, and temporal locality.

Although the examined system provides the concrete case study, the design pattern may apply more broadly. It requires three properties: the expensive computation can be divided into stable units that do not depend on their surrounding context, each unit's result can be identified by its content under a versioned processing configuration, and the results can be reassembled in their original order.

2 Related Work

The preceding chapter established the practical motivation for reusing prior PDF-processing results: documents are resubmitted and revised, yet every submission is sent to the external OCR service in full. Reusing earlier results draws on several research traditions. Memoization and incremental computation explain when an earlier result may replace a new execution; content-addressed storage and deduplication expose redundancy below file-level names; canonicalization and perceptual hashing define when two representations count as the same; caching studies describe how reuse is distributed over time and over objects; and document-processing research identifies the page as a meaningful unit of expensive transformation. This chapter relates these lines of work and uses them to position the page-level mechanism designed in this thesis.

2.1 Memoization, valid reuse and versioning

Memoization is a conceptual ancestor of computation reuse. Michie’s “memo” functions couple a computational rule with a table of previously obtained values, so that a repeated argument can return a stored answer rather than invoke the rule again [9]. Pugh and Teitelbaum subsequently cast function caching as a basis for incremental computation over related inputs [10]. These works show that equal calls may reuse equal answers but leave two operational questions in a real system: what constitutes an equal call, and how long should its answer be retained?

Later work makes those explicit. Selective memoization allows a programmer to control which data participate in matching, records precise dependencies, and couples reuse with space management [6]. Adaptive functional programming goes further by maintaining a dynamic dependence graph and propagating only the consequences of changed inputs [11]. Together, these approaches make reuse validity explicit through controlled equality tests, dependency tracking, and formal correctness conditions.

The correctness of these approaches is defined relative to the inputs and dependencies represented by the reuse mechanism. For a remote, model-based OCR service, this condition can only be approximated through the inputs and configuration visible to the client. The provider may change the model behind a published identifier, a fallback may route a request to a different model, and the recognition of a page might in principle depend on the pages submitted alongside it. A reuse mechanism for such a service must therefore state which parts of the computation it treats as fixed.

Build and deployment systems meet the same issue at a different scale. The design space presented in *Build Systems à la Carte* includes static and dynamic dependencies, early cutoff, and self-tracking builds [12]. Nix places component variants in store paths derived from cryptographic hashes of their declared build inputs, allowing incompatible versions to coexist without being confused [13]. For the present system, these works motivate a cache key that combines content identity with an explicit processing context. The renderer version and the OCR model are part of the logical identity of a result. In this respect, the proposed key is closer to a Nix derivation identity, which distinguishes outputs produced from different declared build inputs, than to a plain content address.

2.2 Reuse in data-intensive and AI workflows

Nectar identifies derived datasets through the computations that produced them and manages their reuse across recurring datacenter workloads [14]. ReStore reuses stored

MapReduce job outputs, DryadInc reuses identical partition computations or incrementally processes appended data, and HELIX optimizes materialization, reuse, and recomputation across iterations of machine-learning workflows [15, 16, 17]. Although their reuse units differ, all attach reusable outputs to information about the computation that produced them.

Derakhshan et al. select artifacts to materialize under a storage budget and generate reuse plans over recurrent data-science DAGs [18]. Cachew further shows that caching is not automatically beneficial: storing and reading transformed data may fail to improve time when dynamic processing already fills the model input pipeline [19]. A useful cache must avoid more cost than it introduces. Chapter 6 states this requirement as a break-even condition on the average overhead.

Prompt Cache moves reuse inside the model: it precomputes attention states for frequently occurring prompt modules and reuses them across prompts [20]. Because the same module can appear at different positions, reuse is correct only if the system defines explicit module boundaries and assigns each module the positions it will occupy in the composed prompt. The page-level mechanism faces a similar requirement at a coarser level. When one cached page result is placed at several positions of a document, its page number and image identifiers must be adapted for each occurrence.

2.3 Content addressing and deduplication

Early storage systems decoupled data identity from pathnames by using content-based addressing. Windows Single Instance Storage detects whole files with identical contents and replaces them with links [21]. Venti uses a cryptographic hash of each block as its address, so that duplicate blocks are stored only once [7]. Foundation offers a mode in which a hash locates candidate duplicates but a byte comparison protects correctness from collisions [22]. These systems separate a logical occurrence from its physical representation, a distinction also required here.

Rsync combines a rolling checksum with a stronger digest to locate reusable fixed-size blocks at arbitrary byte offsets [23]. LBFS instead uses content-defined chunk boundaries so that insertions and deletions affect only nearby chunks [24]. However, byte-level redundancy does not necessarily translate into effective computational reuse, because application services such as OCR require semantically aligned boundaries.

Large-scale deduplication studies quantify this granularity trade-off. REBL combines compression, duplicate-block suppression, and delta encoding to cover multiple forms of redundancy, while Policroniades and Pratt compare whole-file hashing, fixed-size blocks, and content-defined chunks across storage workloads [25, 26]. Meyer and Bolosky find that whole-file deduplication captures a large share of the block-level opportunity in live file systems and an even larger share in backups, but not all of it [27]. Sub-file methods find more reuse, yet their indexes, random lookups, and fragmented layouts create costs of their own.

Production designs therefore optimize granularity together with metadata and I/O. The Data Domain system uses summary structures and locality to prevent bottlenecks; a study shows that larger chunks combined with compression can keep much of the savings of small chunks while substantially reducing metadata; and FastCDC accelerates content-defined chunking while maintaining a deduplication ratio close to Rabin-based schemes [28, 29, 30]. The appropriate unit recognizes valuable redundancy while remaining cheap to identify, store, retrieve, and recombine.

2.4 Canonicalization and hash-based identity

Hash functions can serve two different roles in content-based systems. Rabin fingerprints are efficiently updatable short tags with a controllable collision probability, which makes them suitable for rolling-window matching and content-defined chunking [31]. Page-level reuse does not need this role, because unit boundaries are given by the page structure of the PDF. The question is not where a unit begins but when two units count as the same. Cryptographic hash functions answer this question for byte sequences. Rogaway and Shrimpton distinguish collision resistance, preimage resistance, and second-preimage resistance [32], and NIST standardizes the SHA family of fixed-length message digests [8]. Collision resistance does not make digest equality a proof of byte equality; it makes finding distinct inputs with the same digest computationally infeasible under the security assumption.

Byte equality is often stricter than the equivalence an application needs. Content that is equivalent for the application can appear in different physical representations. *Canonicalization* addresses this by mapping every member of an equivalence class to a single canonical representation before comparison or hashing.

Perceptual image hashing takes a different route to tolerance. Venkatesan et al. compress an image into a short random binary string using randomized signal processing, with robustness against compression and geometric distortions [33]. Monga and Evans derive hashes from prominent image features. Their evaluation illustrates the robustness–discrimination trade-off. Greater invariance to benign transformations can reduce sensitivity to content-altering changes [34]. Perceptual hashes are compared by distance: two images count as the same if their hashes differ by less than a chosen threshold. For OCR reuse, what matters is whether the recognized text is the same, not whether the pages look alike.

Perceptual hashes can be manipulated in both directions. Earlier attacks alter the semantic content of an image while keeping its perceptual hash unchanged, and Hao et al. conversely generate images with only small perceptual changes whose hashes exceed the matching threshold [35]. Struppek et al. show for the deep perceptual hash Neural-Hash that slight image changes can both force and prevent hash collisions [36]. In a cache shared between users, a forced collision would amount to cache poisoning. The stored result of one page would be served for another. The mechanism in this thesis instead combines deterministic canonicalization with cryptographic hashing. The canonicalization function removes the representation differences covered by its normalization rules, and SHA-256 then identifies the resulting canonical byte sequence.

The canonical bytes depend on the renderer, the normalization parameters, the image encoder, and the versions of the libraries that implement them. Work on reproducible builds shows that bit-for-bit identical outputs are obtained only when the whole toolchain is controlled [37]. Nix derives identities from all declared inputs automatically. In the present system, this dependence is captured by a manually maintained version identifier in the cache key, so the validity of cached keys depends on updating it whenever any of these components changes.

2.5 Workload locality and cache retention

Whether a cache is meaningful depends on how requests are distributed over objects and over time. A study of a busy news site found that modified objects often changed only slightly, that a small subset was modified repeatedly, and that popularity showed temporal

stability [38]. Skewed popularity means that a small set of objects accounts for a large share of the hits, while most stored objects are rarely or never requested again.

Retention policies leverage these patterns. ARC keeps items that were used either recently or repeatedly. Adapts to changing access patterns without manual tuning, and does not let items used only once crowd out useful ones [39]. Such policies presuppose knowledge of how quickly and how often objects are reused. The present cache has no eviction policy yet; Chapter 6 measures the concentration of reuse (Section 6.2.3) and its temporal locality (Section 6.2.4) as the empirical basis on which such a policy could later be calibrated.

2.6 Document similarity and PDF processing

For document comparison, Broder formalizes document resemblance and containment using shingle sets and compact sketches, and Williams applies shingling and SimHash to detect near-duplicate papers in an academic digital library [40, 41]. These methods are well suited to clustering and version discovery. They decide whether two documents are sufficiently similar under a threshold, and they operate on whole documents. The mechanism in this thesis tolerates only the variation absorbed by canonicalization and decides equality page by page.

PDF encodes drawing and printing operations rather than explicit document structure. Livathinos et al. classify low-level PDF printing commands with recurrent neural networks to recover document structure, showing why robust conversion requires more than extracting embedded character strings [4]. Nougat instead converts rendered scientific-document pages into markup using a visual Transformer [42]. Because it operates on individual page images, it is compatible with page-level reuse. PaperMage provides a unified representation for composing text- and vision-based models over visually rich scientific documents, while OmniDocBench evaluates parsers across diverse documents and levels [43, 44]. These works primarily target parsing accuracy and structural reliability. In the system studied here, such a parser is an external service whose cost grows with the number of submitted pages. Page-level reuse leaves its output unchanged and reduces only how often it is called.

2.7 Summary

Table 2.1 summarizes the unit, the domain, and the main difference of representative prior work with respect to the setting of this thesis.

Taken together, prior work provides principles for valid computation reuse, content-derived identity, canonical equivalence, and page-oriented document transformation. The contribution of this thesis is to integrate these established ideas at an existing page-processing boundary, using deterministic canonicalization followed by exact hashing as the page identity, and to evaluate whether exact reuse reduces repeated OCR work in a deployed submission workflow.

Work	Unit	Domain	Difference from this setting
<i>Computation reuse (unit that is reused)</i>			
Memo functions, selective memoization [9, 6]	Function call with matched arguments	Programming languages	Does not address persistent result sharing across users or submissions.
Adaptive functional programming [11]	Nodes of a dynamic dependence graph	Incremental computation	Requires dependencies to be exposed and tracked; dependencies internal to an opaque service are unavailable.
Build systems à la carte, Nix [12, 13]	Build task, component derivation	Build and deployment	Run builds locally and hash the exact declared inputs.
Nectar, ReStore, DryadInc [14, 15, 16]	Derived dataset, job output, partition	Datacenter dataflow	Requires integration with the computation graph and its lineage.
HELIX, Derakhshan et al. [17, 18]	Intermediate workflow artifacts	ML and data science pipelines	Requires a known workflow DAG and control over materialization within the workflow system.
Cachew [19]	Transformed input dataset	ML input pipelines	Reuses whole preprocessed datasets identified by the pipeline definition, not individual units identified by content.
Prompt Cache [20]	Attention states of prompt modules	LLM inference	Requires server-side access to attention states and positional metadata.
<i>Storage deduplication and chunking (unit that is compared)</i>			
SIS, Venti, Foundation [21, 7, 22]	Whole file, block	Storage systems	Deduplicate stored data by its raw bytes; do not canonicalize equivalent representations or reuse computed results.
Rsync, LBFS, FastCDC [23, 24, 30]	Fixed-size receiver block or content-defined chunk	File transfer and storage	Byte-level boundaries are not aligned with pages as units of OCR.

Table 2.1: Representative related work grouped by its role for this thesis.

3 Problem Formulation and Reuse Algorithms

At a broad level, successive inputs may be different as complete objects but share large portions of content whose processing semantics have not changed. Treating every new object as entirely new causes computational cost to grow with total input size, failing to reflect the actual amount of new information being analyzed.

This motivates fine-grained computation reuse: decompose a structured input into reusable units, retain previously computed results by content-derived identities, and repeat expensive processing only for units with no valid result. Incrementally revised PDF documents provide a concrete instance of this general problem. Different submissions, or different versions of the same submission, may contain many identical pages even though their document-level identities differ.

In the system studied in this thesis, PDFs are submitted to an OCR service that charges per page, and the service returns page-level results that are later assembled into a document-level Markdown representation. Therefore, safely reusing computation across partially repeated inputs can be framed as a page-level problem.

3.1 Problem Statement and Scope

The objective is to make the amount of expensive processing depend on content that is new compared to the available result cache.

Let documents arrive as a sequence $D_1, D_2, \dots$. Document D_t is an ordered sequence of page occurrences

$$D_t = (x_{t,1}, x_{t,2}, \dots, x_{t,n_t}) \quad (3.1)$$

where n_t is the number of pages and $x_{t,i}$ is the page occurring at position i . The distinction between a page *occurrence* and page *content* is important: identical content can occur at several positions within one document or across different submissions.

Let $\mathcal{X}$ denote the space of input PDF pages. Let $f_\theta : \mathcal{X} \rightarrow \mathcal{R}$ denote the page-local OCR processor under OCR model identifier θ . For page $x_{t,i}$, let

$$r_{t,i} = f_\theta(x_{t,i}) \quad (3.2)$$

denote the reusable OCR output before document-specific page metadata is assigned. Let

$$z_{t,i} = \text{AdaptPage}(r_{t,i}, i) \quad (3.3)$$

denote the page result adapted to position i in document D_t . The complete output is then

$$Y_{\text{full}}(D_t) = \text{Assemble}(z_{t,1}, z_{t,2}, \dots, z_{t,n_t}) \quad (3.4)$$

The formal problem can be stated as follows: given D_t , f_θ , a persistent cache of earlier results, and an output assembly function `Assemble`, retrieve results for matching page keys and process the remaining distinct pages to reduce the number of pages submitted to OCR. Then reconstruct the complete result in the original page order.

3.1.1 Content Identity and Cache Validity

Raw PDF objects are not suitable cache identities because visually equivalent pages can have different encodings. The versioned canonicalization function is

$$\phi_v : \mathcal{X} \rightarrow \{0, 1\}^*. \quad (3.5)$$

For page $x_{t,i}$, define

$$b_{t,i} = \phi_v(x_{t,i}), \quad (3.6)$$

$$h_{t,i} = \text{Hash}(b_{t,i}). \quad (3.7)$$

For any page x , define the cache-key function

$$\kappa_{v,\theta}(x) = (\text{Hash}(\phi_v(x)), v, \theta). \quad (3.8)$$

For page occurrence $x_{t,i}$, its cache key is therefore

$$\kappa_{t,i} := \kappa_{v,\theta}(x_{t,i}) = (h_{t,i}, v, \theta). \quad (3.9)$$

The key consists of the page hash result, the canonicalization version, which records the rendering and normalization procedure, and the OCR model identifier. Including v and θ prevents entries created using different recorded preprocessing versions or OCR models from sharing the same key. This separation also allows results produced under different recorded rendering configurations and OCR models to be distinguished during subsequent analysis.

Let

$$\mathbb{K} = \text{Hash} \times \mathcal{V} \times \Theta$$

denote the cache-key space, where $\mathcal{V}$ is the set of canonicalization versions and Θ is the set of recorded OCR model identifiers. Other output-relevant OCR options are assumed to remain fixed, and every OCR request is assumed to be served by the configured model θ .¹

The persistent cache is a partial mapping

$$\mathcal{C}_{t-1} : \mathbb{K} \rightharpoonup \mathcal{R}. \quad (3.10)$$

An entry is treated as a hit only when this complete key exists and references a retrievable stored OCR result. Local-failure placeholders are never stored as cache entries.

Let $\simeq_\theta$ denote equivalence of page-level OCR results for the downstream task. This relation expresses that two results are interchangeable for the intended use and does not necessarily require byte-for-byte equality.

For reuse to preserve the intended page-local result, equality of cache keys is required to be processing-preserving under the same recorded configuration:

$$\kappa_{v,\theta}(x) = \kappa_{v,\theta}(x') \implies f_\theta(x) \simeq_\theta f_\theta(x'). \quad (3.11)$$

Equation 3.11 is a system-level validity assumption. Its interpretation depends on both the canonicalization function and the hash function.

¹The deployed OCR client can fall back to an alternative model after transient failures. Such results are stored under the identifier of the model that produced them, so individual cache entries remain consistent, but a document may then combine results from two models. This case is outside the scope of this thesis.

The canonicalization function is not required to be injective. Distinct source pages may intentionally be mapped to the same canonical byte sequence when their differences are considered irrelevant to page-level OCR for the target document domain. This constitutes the intended equivalence relation introduced by canonicalization. The corresponding semantic assumption is

$$\phi_v(x) = \phi_v(x') \implies f_\theta(x) \simeq_\theta f_\theta(x').$$

On the hash side, Hash is instantiated as SHA-256 and treated as collision-resistant, so that equal digests are taken to imply equal canonical byte sequences. Under this assumption, Equation 3.11 reduces to the semantic assumption on ϕ_v stated above.

The design further assumes *page locality*: the OCR result for a page depends only on that page and not on neighboring pages or document-wide context. Processing a page within a reduced document containing a subset of the original pages yields a result equivalent to processing it inside the original document.

3.1.2 Workload Quantities

Define the set of distinct page keys generated from the submitted document and its size by

$$\mathcal{K}_t = \{\kappa_{t,i} \mid 1 \leq i \leq n_t\}, \quad d_t = |\mathcal{K}_t|. \quad (3.12)$$

Thus, d_t counts distinct page keys after within-document duplicates have been collapsed.

The set of distinct keys not present in the cache and its size are as follows:

$$\mathcal{Q}_t = \mathcal{K}_t \setminus \text{dom}(\mathcal{C}_{t-1}), \quad m_t = |\mathcal{Q}_t|. \quad (3.13)$$

To distinguish distinct missing keys from their occurrences in the original document, define

$$I_t^{\text{miss}} = \{i \mid \kappa_{t,i} \in \mathcal{Q}_t\}, \quad \ell_t = |I_t^{\text{miss}}|. \quad (3.14)$$

Let $Q_t = (q_{t,1}, \dots, q_{t,m_t})$ be the ordered enumeration of $\mathcal{Q}_t$, sorted by the position of the first corresponding page occurrence.

Here, ℓ_t counts cache-miss page occurrences, whereas m_t counts distinct missing keys. Because repeated occurrences of the same missing key are represented by a single page before OCR, m_t is the number of representative pages that must be newly processed. The quantities therefore satisfy

$$m_t \leq d_t \leq n_t, \quad m_t \leq \ell_t \leq n_t.$$

No fixed ordering exists between d_t and ℓ_t , because d_t includes both cached and cache-missing distinct keys, whereas ℓ_t counts all occurrences whose keys are missing.

The model separates two sources of avoided processing:

$$\rho_t^{\text{cache}} = \frac{n_t - \ell_t}{n_t}, \quad \text{reuse from the persistent cache,} \quad (3.15)$$

$$\rho_t^{\text{dup}} = \frac{\ell_t - m_t}{n_t}, \quad \text{deduplication among missing occurrences,} \quad (3.16)$$

$$\rho_t^{\text{eff}} = 1 - \frac{m_t}{n_t} = \rho_t^{\text{cache}} + \rho_t^{\text{dup}}, \quad \text{total avoided-processing fraction.} \quad (3.17)$$

These rates characterize an individual submission and count page occurrences. The persistent counter *hitCount* of a cache entry is instead incremented once per matching index

record for each distinct key. Under the normal processing path, submission t therefore contributes $d_t - m_t \leq n_t - \ell_t$ to the total. A cumulative hit rate derived from these counters is therefore a distinct-key rate and may underestimate ρ_t^{cache} when cached pages repeat within a document.

3.1.3 Cost Model and Optimization Objective

Let c_f denote the per-page OCR cost in a cost dimension for which processing cost is approximated as linear in the number of pages, such as the provider's page-based charge. Let c_{id} be the per-page cost of rendering, canonicalization, and hashing; $C_{\text{lookup}}(d_t)$ the cost of querying the cache index for d_t distinct keys and retrieving and loading the stored OCR results referenced by valid entries; $C_{\text{pack}}(m_t)$ the cost of constructing the temporary PDF; $C_{\text{store}}(m_t)$ the cost of storing newly produced results; and $C_{\text{merge}}(n_t)$ the additional cost of mapping cached and new results to their original positions before document assembly. Terms may represent money, latency, or compute, but all terms in one application of the model must use the same unit. Ignoring work shared by both processing paths, processing the entire document has cost

$$C_{\text{full}}(D_t) = n_t c_f. \quad (3.18)$$

The page-level reuse design has cost

$$\begin{aligned} C_{\text{reuse}}(D_t) = & m_t c_f + n_t c_{id} + C_{\text{lookup}}(d_t) \\ & + C_{\text{pack}}(m_t) + C_{\text{store}}(m_t) + C_{\text{merge}}(n_t). \end{aligned} \quad (3.19)$$

Define the reuse overhead

$$O_t = n_t c_{id} + C_{\text{lookup}}(d_t) + C_{\text{pack}}(m_t) + C_{\text{store}}(m_t) + C_{\text{merge}}(n_t). \quad (3.20)$$

The net saving predicted by the model is therefore

$$\Delta C_t = (n_t - m_t) c_f - O_t, \quad (3.21)$$

and the design is beneficial in the selected cost dimension when

$$(n_t - m_t) c_f > O_t. \quad (3.22)$$

Equivalently,

$$\rho_t^{\text{eff}} > \frac{O_t}{n_t c_f}. \quad (3.23)$$

Cache storage is reported separately because bytes cannot be added directly to monetary or latency costs. Let

$$S_C = S_{\text{data}} + S_{\text{index}}, \quad (3.24)$$

denote the observed storage occupied by cache documents and their indexes.

Because the OCR provider charges per page, the primary operational objective is to reduce the number of pages submitted for OCR. Without reuse, n_t pages are processed; with reuse, only the m_t distinct cache-miss pages are processed. The reconstructed result must nevertheless preserve the page order of the original document. The implemented unit of processing and reuse is one PDF page.

The monetary OCR cost can be approximated as linear in the number of processed pages. However, latency and computational cost may not decrease proportionally with the page count because each request has a fixed overhead and the OCR service processes pages in batches.

3.2 Reuse Algorithms

Within this section, the submission index t is omitted when only the current document is considered: $x_i = x_{t,i}$, $\kappa_i = \kappa_{t,i}$, $n = n_t$, $m = m_t$, $\ell = \ell_t$, $Q = Q_t$, and $\mathcal{C} = \mathcal{C}_{t-1}$.

The mechanism is formalized as two main algorithms. Algorithm 1 performs canonicalization, key generation, cache lookup, and missing-page identification. Algorithm 2 processes the identified cache misses and reconstructs the result in the order of the original document.

For each input page x_i , Algorithm 1 first applies the versioned canonicalization function ϕ_v and constructs a cache key κ_i from the resulting hash, the canonicalization version v , and the OCR model identifier θ .

The representative map Rep records the first occurrence of each distinct key. Each distinct key is looked up only once. If the key is missing, its first occurrence is used as the representative page for OCR processing. The complete sequence $(\kappa_1, \dots, \kappa_n)$ is retained so that the result associated with each key can later be assigned to every corresponding page occurrence in its original position.

Algorithm 1 Cache Lookup and Missing-Page Identification

Require: Document pages $D = (x_1, \dots, x_n)$, OCR model identifier θ , canonicalization version v , and cache $\mathcal{C}$

Ensure: Page keys $(\kappa_1, \dots, \kappa_n)$, representative map Rep , cached results $Result$, and ordered missing-key sequence Q

```

1:  $Rep \leftarrow \emptyset$ 
2: for  $i \leftarrow 1$  to  $n$  do
3:    $\kappa_i \leftarrow (\text{Hash}(\phi_v(x_i)), v, \theta)$ 
4:   if  $\kappa_i \notin \text{dom}(Rep)$  then
5:      $Rep[\kappa_i] \leftarrow i$ 
6:   end if
7: end for
8:  $Keys \leftarrow [\kappa \in \text{dom}(Rep) \mid \kappa \text{ ordered by increasing } Rep[\kappa]]$ 
9:  $Result \leftarrow \text{LookupValid}(\mathcal{C}, Keys)$ 
10:  $Q \leftarrow [\kappa \in Keys \mid \kappa \notin \text{dom}(Result)]$ 
11: return  $(\kappa_1, \dots, \kappa_n), Rep, Result, Q$ 

```

Algorithm 1 returns the unresolved distinct keys as an ordered sequence. The missing-key sequence Q follows the original document order, as determined by the representative position $Rep[\kappa]$ of each key. If all distinct keys have valid cache entries, then Q is empty, and the OCR operation can be skipped entirely.

When cache misses exist, Algorithm 2 constructs a temporary PDF D^{miss} containing one representative page for each distinct missing key, so that its page count is $m = |Q|$.

Let

$$\text{BatchOCR}_{\theta}(D^{\text{miss}}) = (w_1, \dots, w_s)$$

denote the ordered sequence returned by a single external OCR request for the temporary PDF D^{miss} . Here, s is the number of returned page-level results.

The representative pages are inserted into D^{miss} in the order of the missing-key sequence $Q = (q_1, \dots, q_m)$. In the complete, order-preserving case, and under the page-locality

assumption, the response satisfies

$$s = m, \quad w_j = f_\theta(x_{Rep[q_j]}), \quad 1 \leq j \leq m.$$

The implementation relies on the following response-order assumption: for every $j \leq \min(s, m)$, the j -th returned result corresponds to the j -th page of D^{miss} . Under this assumption, results $w_1, \dots, w_{\min(s, m)}$ are associated positionally with $q_1, \dots, q_{\min(s, m)}$. If $s < m$, the remaining keys $q_{s+1}, \dots, q_m$ receive no result, and their page occurrences are represented by $\text{LocalFailure}(i)$, an empty placeholder that is not stored in the persistent cache. If $s > m$, results beyond w_m are discarded. The count s is therefore not used to validate the correspondence.

Any page number contained in w_j is treated as output metadata and is not used as the mapping key. During AdaptPage , this metadata is replaced by the corresponding position in the current original document.

Finally, the algorithm iterates over the original key sequence $(\kappa_1, \dots, \kappa_n)$. Cached and newly computed results are handled uniformly through the key-indexed result map. For each available result, AdaptPage creates the document-specific page result for its current position. The adapted page results are then assembled as $(z_1, \dots, z_n)$.

Together, the two algorithms ensure that the number of pages submitted to the external processor is determined by the number of distinct unresolved keys.

Algorithm 2 Missing-Page Processing and Result Assembly

Require: Document pages $D = (x_1, \dots, x_n)$, page keys $(\kappa_1, \dots, \kappa_n)$, representative map Rep , result map $Result$ initially containing the valid cache hits, ordered missing keys $Q = (q_1, \dots, q_m)$, OCR model identifier θ , and cache $\mathcal{C}$

Ensure: Ordered document-level result $Y_{\text{reuse}}(D)$

```

1: if  $m > 0$  then
2:    $D^{\text{miss}} \leftarrow \text{BuildTemporaryPDF}(x_{Rep[q_1]}, \dots, x_{Rep[q_m]})$ 
3:    $(w_1, \dots, w_s) \leftarrow \text{BatchOCR}_\theta(D^{\text{miss}})$ 
4:   for  $j \leftarrow 1$  to  $\min(s, m)$  do
5:      $Result[q_j] \leftarrow w_j$ 
6:      $\text{Store}(\mathcal{C}, q_j, w_j)$ 
7:   end for
8: end if
9: for  $i \leftarrow 1$  to  $n$  do
10:  if  $\kappa_i \in \text{dom}(Result)$  then
11:     $z_i \leftarrow \text{AdaptPage}(Result[\kappa_i], i)$ 
12:  else
13:     $z_i \leftarrow \text{LocalFailure}(i)$ 
14:  end if
15: end for
16:  $Y_{\text{reuse}}(D) \leftarrow \text{Assemble}(z_1, \dots, z_n)$ 
17: return  $Y_{\text{reuse}}(D)$ 

```

Provided that the batch response is complete ($s = m$) and preserves the temporary-PDF page order, and provided that page locality and Equation 3.11 hold, the reconstructed output is semantically equivalent to full processing:

$$Y_{\text{reuse}}(D_t) \simeq Y_{\text{full}}(D_t). \quad (3.25)$$

The equivalence claim does not cover incomplete or reordered responses. If $s < m$, pages associated with $q_{s+1}, \dots, q_m$ are reported as local failures. If the service omits or reorders a page other than at the end of the response, the positional association becomes misaligned and incorrect results may be stored in the persistent cache. The implementation does not detect this case and relies on the ordering behavior of the OCR service. Likewise, returned results are stored without further content validation.

4 Page-Level Reuse for PDF Processing

This chapter describes how the reuse model of Chapter 3 is realized in the document-processing pipeline of the AI-assisted review system. The notation of that chapter is used throughout, with the submission index t suppressed. Page positions in formulas are one-based, whereas the implementation uses zero-based page indices (Section 5.2).

4.1 Baseline Full-Document OCR Workflow

Before the introduction of page-level reuse, each uploaded PDF was submitted in its entirety to the Mistral OCR API, even though the OCR response contained page-level results. As illustrated in Figure 4.1, the pipeline consisted of PDF submission, content extraction, and persistence of the resulting artifacts. Its purpose was to transform an uploaded paper into machine-readable content for subsequent stages of the review system.

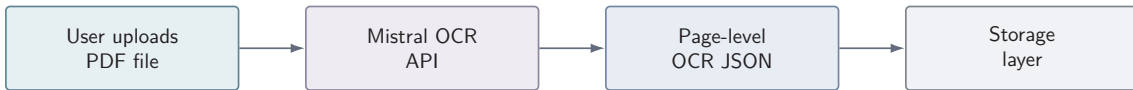


Figure 4.1: Original workflow

Mistral OCR accepts PDFs and images and extracts textual and visual content, which makes it applicable to scientific papers containing text, figures, tables, and mathematical expressions. Its response is a JSON object containing page-level entries, each with a page index, Markdown content, and image information when images are extracted. The source PDF and the derived JSON, Markdown, and image outputs are then retained through the storage layer for subsequent application components.

The principal limitation of this workflow is that it performs no cache lookup before invoking OCR, so uploading an identical document again, or a revised version with only a few changed pages, leads to OCR processing of content that has already been processed. This workflow corresponds to the full-processing baseline C_{full} of Equation 3.18.

4.2 Mechanism Design

4.2.1 Motivation for Page-Level Processing

The page-level design was selected for four main reasons.

First, Mistral OCR is priced per page, so the cost is proportional to the number of pages submitted. If only two pages of a twenty-page paper have changed and the remaining eighteen pages match valid cache entries, submitting only the two unresolved pages avoids the OCR charge for the other eighteen.

Second, Mistral OCR already represents its output as a list of page objects. Page-level caching therefore corresponds directly to the provider’s response structure and does not require a continuous document-level output to be divided after processing.

Third, page-level reuse supports revised submissions. Unchanged pages can be reused even when other pages have been inserted, deleted, or modified, because cached results are not tied to the position at which they were first produced.

Fourth, page-level reuse also handles duplicate pages within one PDF: only the first occurrence of a missing page is submitted to OCR, and its result is reused for every occurrence.

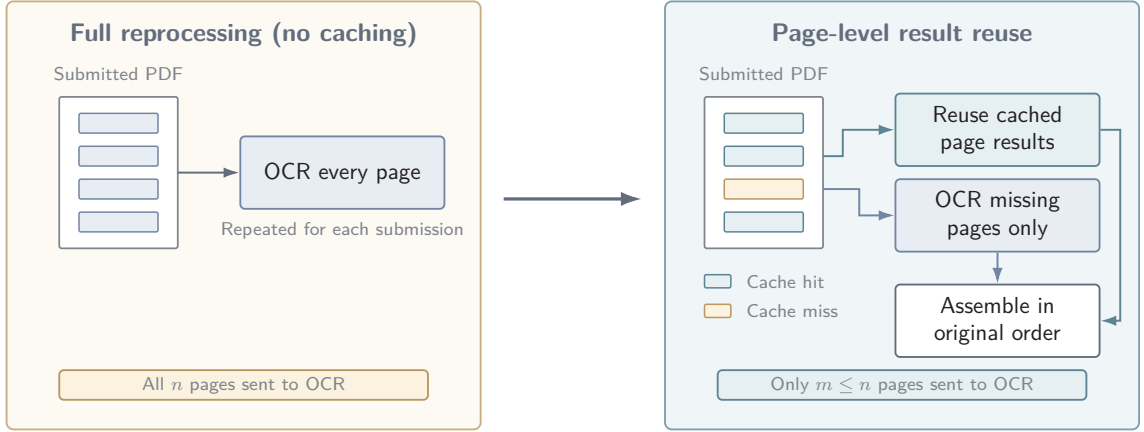


Figure 4.2: From full-document reprocessing to page-level result reuse

As shown in Figure 4.2, instead of reprocessing an entire document on every request, the system recomputes only pages without a valid cached result and reuses the results of all other pages.

The mechanism consists of three stages. Initial processing computes the page keys; hash-based page matching realizes Algorithm 1 and the OCR part of Algorithm 2; and result integration realizes the assembly part of Algorithm 2.

4.2.2 Initial Processing

Initial processing establishes the identity of the submission and produces the page keys required for cache lookup. As illustrated in Figure 4.3, it comprises five steps: uploading the PDF, generating a paper ID, rendering its pages as images, canonicalizing the rendered images, and computing a hash for each canonical page representation. Its output is the ordered sequence of page hashes $(h_1, \dots, h_n)$ together with the original page positions.



Figure 4.3: Initial page-level processing workflow

The paper ID identifies the submission record and is used to associate the source document with its outputs.¹ The page hash, in contrast, identifies page content across records: submissions with different paper IDs can contain pages with matching hashes, which allows results to be reused across submissions while outputs remain associated with the submission that requested them.

Each page is rendered as a PNG image, and the canonicalization function ϕ_v is applied to this rendering; h_i is the SHA-256 digest of the result. The rendered image is used only for key generation and is not submitted to OCR. Because resolution, color space, and preprocessing affect the canonical representation, these settings are fixed for a given canonicalization version v . The concrete rendering and canonicalization procedure is described in Section 5.2.

¹In the implementation, the paper ID is the review identifier created when the submission is registered. It is also recorded in the `relatedReviewIds` field of every cache entry that the submission creates or reuses.

4.2.3 Hash-Based Page Matching

Hash-based page matching determines which pages require OCR, as illustrated in Figure 4.4. It uses the cache keys $\kappa_i = (h_i, v, \theta)$ of Equation 3.9 to identify reusable results, selects one representative page for each distinct missing key, and restricts new OCR work to those pages.

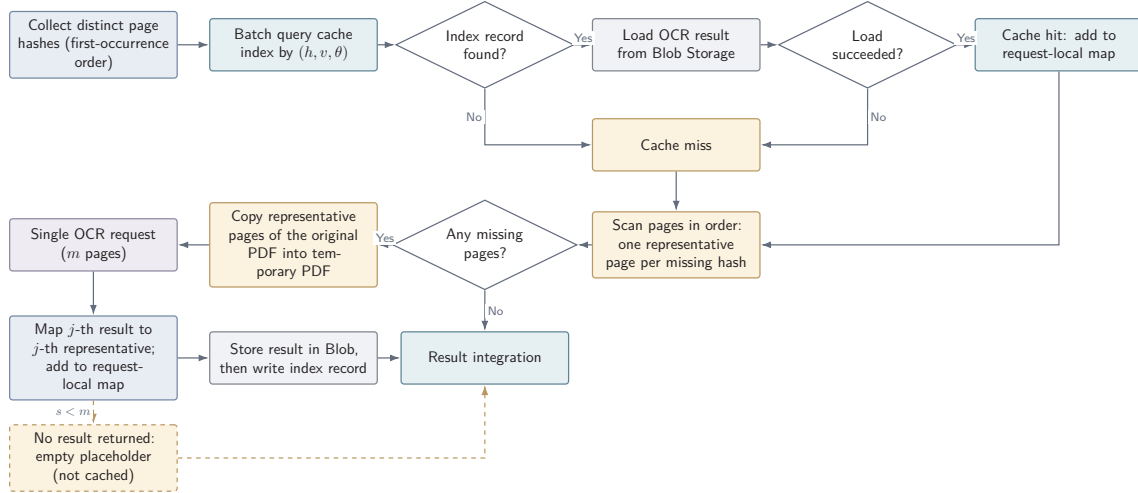


Figure 4.4: Workflow of hash-based page matching

The design applies content-based identification at page granularity. A related principle appears in Venti, where content fingerprints identify stored blocks and allow duplicate blocks to share a stored representation [7]. Here, the hash identifies a canonical page representation and provides access to a previously computed OCR result. Reuse therefore depends on page content rather than on the uploaded filename or the page’s position in an earlier document.

Cache lookup and identification of missing pages: The distinct page hashes are collected in order of first occurrence and looked up in a single batch query against the cache index, restricted to the current canonicalization version v and configured OCR model θ . A matching index record counts as a hit only if the referenced OCR result can also be loaded from Blob Storage; otherwise the page is treated as a miss. This realizes `LookupValid` in Algorithm 1. Loaded results are placed in a request-local map indexed by page hash.

The ordered page list is then scanned. A page whose hash is absent from the request-local map and has not already been selected, becomes the representative page of its key. Later occurrences of the same hash do not create additional representatives. The resulting representative list corresponds to Rep and the ordered missing-key sequence $Q = (q_1, \dots, q_m)$.

Selective OCR using a temporary PDF: If Q is empty, temporary-PDF creation and OCR are skipped. Otherwise, the pages at positions $Rep[q_1], \dots, Rep[q_m]$ are copied from the original PDF into a temporary PDF D^{miss} , which is submitted in a single OCR request. The returned results are associated positionally with $q_1, \dots, q_m$, under the response-order assumption and with the handling of incomplete responses stated in Chapter 3.

For example, suppose that positions 2 and 5 of a six-page submission contain two different unresolved keys and that no other positions are unresolved. The temporary PDF then contains these two pages in the same order, and its first and second OCR results are mapped to original positions 2 and 5. If positions 2 and 5 instead share the same

unresolved key, only the page at position 2 is included, and its OCR result is reused at both positions.

The temporary document is a processing artifact; the original document remains the reference for page ordering and output reconstruction.

Persistence of new results: Each new result is inserted into the request-local map under its representative page hash and is then persisted in two steps: the OCR result is first written to the dedicated `page-ocr-cache` Blob Storage container, and only then is the corresponding index record written. Publishing the index record last ensures that a lookup never returns a reference to content that has not yet been stored. Each write is retried up to three times with increasing back-off. If the retries are exhausted, processing of the submission fails and is retried by the task queue; because the blob address and index key are deterministic, repeating the persistence step is idempotent. Local-failure placeholders are never persisted.

By construction, m representative pages are submitted to OCR instead of n , which yields the avoided-processing fraction ρ^{eff} of Equation 3.17. This fraction measures avoided OCR page work rather than end-to-end speedup, since the overhead terms of Equation 3.19 remain.

4.2.4 Result Integration

Result integration realizes the final loop of Algorithm 2, as illustrated in Figure 4.5. At this point, the request-local map contains both the cache hits loaded during lookup and the newly produced OCR results, so no result is reloaded from Blob Storage.

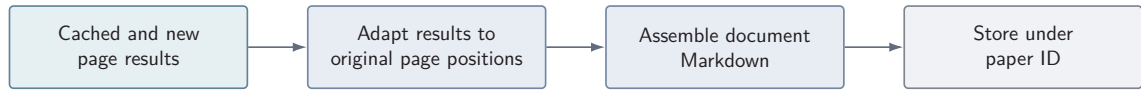


Figure 4.5: Workflow of result integration

Adapting results to page positions: The pages are traversed in their original document order. For each page x_i , the result stored under h_i is copied and adapted to position i , which realizes `AdaptPage`: the page identifier, page number, page hash, and canonicalization version are replaced, and each extracted-image identifier is prefixed with the one-based position of the page (for example, `page-3-img-0.jpeg`), with the corresponding image references in the Markdown rewritten accordingly. This traversal interleaves cache hits and new results in document order. When two positions share a hash, both retrieve the same OCR content, but adaptation produces separate page-specific objects, so image identifiers remain unique within the document.

Generating the document-level Markdown: The Markdown of each adapted page is preceded by a marker of the form `--- Page  $i$  ---`, and the pages are concatenated in document order to produce a single document-level representation. This realizes `Assemble`. The resulting representation and the page-level OCR results are stored under the paper ID for subsequent stages of the review system.

5 Production Implementation

This chapter describes how the design of Chapter 4 is implemented in the production code base. It first summarizes the original workflow. Then follows the page-level reuse path in execution order and closes with the validation of the implementation. Page indices in this chapter are zero-based, as in the code.

5.1 Implementation of the Original Workflow

Before page-level reuse was introduced, every uploaded PDF was processed as one complete unit. The workflow has two phases. A synchronous registration phase validates and stores the upload. An asynchronous background phase performs OCR, generates Markdown, extracts metadata, and persists the results.

5.1.1 Registration and Task Dispatch

An authenticated user submits the PDF and the review configuration as multipart form data. The endpoint delegates the request to `RegisterHandler.handle_pdf()`. This method reads the whole PDF into memory and validates it. The file may not exceed 20 MiB. The page count may not exceed the user's individual limit or the configurable system default.

After validation, the handler creates a review record in MongoDB with the status *pending*. Its identifier serves as the namespace for all storage artifacts of the request. The original PDF is then stored in Azure Blob Storage, and the review is charged. Finally, the review identifier is placed on the `review-tasks` Azure Storage Queue, and the API returns HTTP 202 Accepted.

5.1.2 Review-Specific Artifact Lookup

A background worker picks up the task and calls `ReviewEngine.run_pdf()`. Before any OCR, the engine calls `PDFIngestionService.retrieve_if_exists()`. This method checks whether an extraction already exists at the storage location of this review. If it does, the stored Markdown, references, and images are loaded, and OCR is skipped. Otherwise, the complete PDF proceeds to OCR.

This check only prevents repeated work within the same review record. It cannot reuse results across different uploads.

5.1.3 Full-Document OCR Processing

`PDFIngestionService.process_ocr_with_metadata()` downloads the original PDF from Blob Storage and passes it to `OCRService`. The external call is delegated to `MistralAIClient`.

The client Base64-encodes the entire PDF as an inline document URL. It submits this URL to the Mistral OCR API with the configured model, by default `mistral-ocr-latest`. Image extraction is enabled through the `include_image_base64` option. One request thus covers all pages of the document.

Mistral returns one entry per page, with Markdown text and detected images. The client converts each entry into an internal `OCRChop` object with three fields: `markdown`, `page_number`, and `images`. Provider-specific fields are dropped. During this conversion, the Markdown is processed by `neutralize_html_artifacts()`. This function neutralizes suspected malformed HTML-like content introduced by OCR. Legitimate HTML structures, such as tables, are retained.

Mistral sometimes rejects a file as an invalid PDF. In that case, a fallback applies. Ghostscript rewrites the document as PDF 1.6 and downsamples its images to 150 DPI. The normalized PDF is stored next to the original upload, and OCR is retried once.

5.1.4 Markdown Synthesis

`OCRService.extract_markdown()` joins the page results into one document-level Markdown string. Pages are joined in the order returned by Mistral. A marker `--- Page N` precedes each page. These markers keep page boundaries visible. Later steps rely on them, for example, to select the first two pages for metadata extraction.

5.1.5 Metadata Extraction

Next, the system counts the pages and the OCR-detected images. It sends the Markdown of the first two pages to a language-model-based extractor, which returns the title, authors, and abstract.

This step runs before the OCR results are persisted. A slow metadata response therefore delays storage, although the Markdown is already in memory.

5.1.6 Artifact Persistence and Reconstruction

The OCR response (`ocr-data.json`) and the document-level Markdown are then written to Blob Storage concurrently. For later processing, `PDFIngestionService.reconstruct()` reloads the stored JSON and rebuilds the Markdown with the same concatenation function. During reconstruction, the reference section is extracted if present. Detected images are decoded, resized, converted to JPEG, and stored as separate artifacts. The result is passed to review generation.

5.1.7 Limitations of the Original Implementation

The original workflow reliably turns a complete PDF into structured OCR data and Markdown. It does not compare content across uploads. Nor does it recognize unchanged pages. A repeated submission, or a revision with minor changes, is therefore sent to the OCR provider in full. This is the full-processing baseline of Equation 3.18.

5.2 Implementation of the Reuse Mechanism

The reuse mechanism changes only the background OCR stage. Upload, validation, charging, and task dispatch remain as described above. Figure 5.1 shows the resulting architecture. Each page is identified independently. Pages with a usable cached result are served from the cache. The remaining pages are copied into one temporary PDF and sent in a single OCR request. Cached and new results are then merged in the page order of the upload. The following subsections follow this path in execution order.

5.2.1 Background Worker and Pipeline Initialization

In the queue-based deployment, the entry point is the `review_processor()` Azure Function. It is triggered by a message on the `review-tasks` queue. The function decodes the message and reads the review identifier. It then initializes the required infrastructure:

- MongoDB for review records and cache metadata;
- Azure Blob Storage for PDFs and OCR artifacts;
- Redis for progress events;
- the Mistral OCR client;
- the PDF ingestion service; and
- the page OCR cache repository.

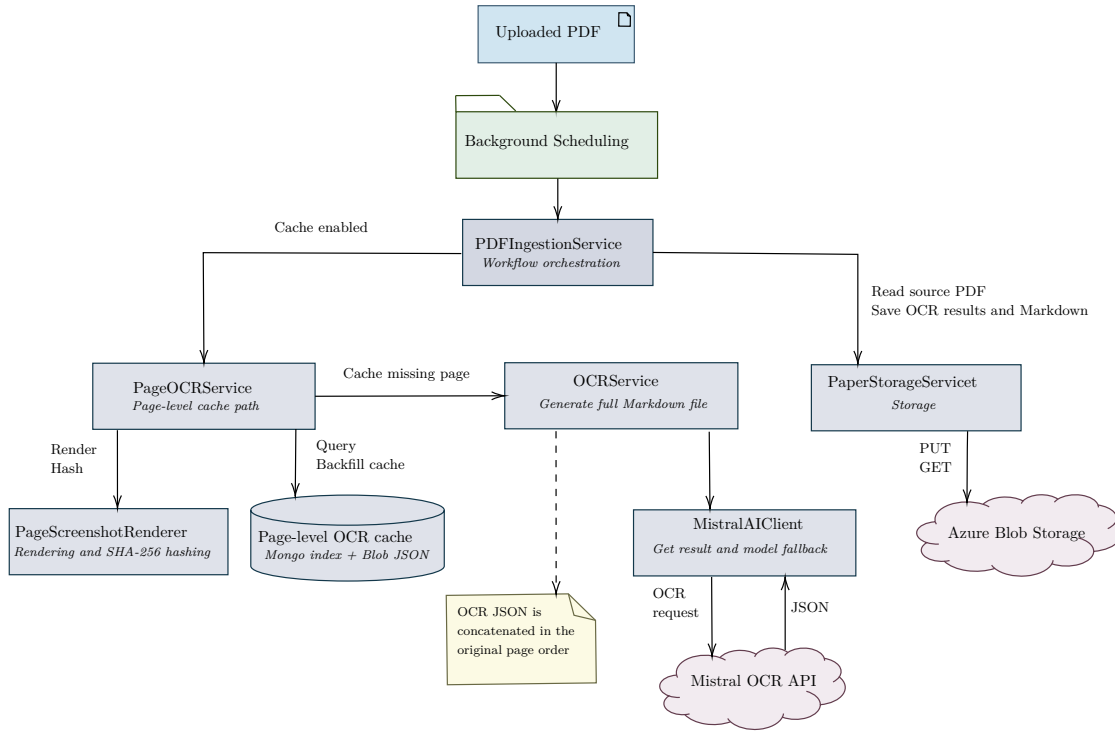


Figure 5.1: Architecture of the page-level reuse workflow.

`ReviewEngine.run_pdf()` loads the review record and checks that it is still eligible. Completed reviews are skipped. The review-specific artifact lookup described above still runs first. Only if it finds nothing does the engine call `PDFIngestionService.process_ocr_with_metadata()`, which now routes OCR through `PageOCRService`.

Figure 5.2 shows how this service turns each page into a hash. The following subsections describe the steps in order. Together they implement the canonicalization function ϕ_v and the hash of Equation 3.9. The version v is the constant `RENDER_VERSION`, which identifies the whole rendering and normalization procedure.



Figure 5.2: Workflow of page image normalization.

5.2.2 Page Rendering

`PageScreenshotRenderer` opens the PDF with `PyMuPDF` and renders the pages in their original order. Each page is rasterized at a scale factor of 2.0 (Listing 5.1).

```

1 import fitz
2 RENDER_SCALE = 2.0
3
4 matrix = fitz.Matrix(RENDER_SCALE, RENDER_SCALE)
5 pixmap = page.get_pixmap(
6     matrix=matrix,
7     alpha=False
8 )
  
```

Listing 5.1: Page rendering

The scale factor doubles the raster size along both axes. The increased raster resolution is intended to preserve small characters, mathematical symbols, and thin lines that may be degraded at lower resolutions. At a lower resolution, some of these details could be lost.

Transparency is disabled. With PyMuPDF's default RGB color space, this yields an opaque three-channel image. Transparent regions are rendered against an explicit background, so alpha handling cannot cause differences.

The page is encoded as PNG. These PNG bytes are not hashed directly. Visually equivalent pages may still differ slightly at pixel level. Such differences come from rasterization, antialiasing, compression, or the source PDFs themselves. The image should be passed through a normalization pipeline first.

5.2.3 Grayscale Conversion

OpenCV decodes the PNG bytes with `cv2.IMREAD_GRAYSCALE` (Listing 5.2). The result has a single grayscale intensity channel.

```
1 image_array = np.frombuffer(image_bytes, dtype=np.uint8)
2 image = cv2.imdecode(image_array, cv2.IMREAD_GRAYSCALE)
```

Listing 5.2: Grayscale conversion

Color no longer contributes to the hash. It therefore reduces the sensitivity of the cache key to color differences between pages with otherwise equivalent text and layout. This behavior is intentional for the target document domain, in which the reusable OCR result is primarily determined by textual and structural content.

5.2.4 CLAHE Contrast Normalization

Then, Contrast-Limited Adaptive Histogram Equalization (CLAHE) is applied with a clip limit of 2.0 and an 8×8 tile grid (Listing 5.3).

```
1 clahe = cv2.createCLAHE(
2     clipLimit=2.0,
3     tileGridSize=(8, 8)
4 )
5
6 image = clahe.apply(image)
```

Listing 5.3: CLAHE contrast normalization

Unlike global histogram equalization, CLAHE works on local regions. It evens out uneven backgrounds, low-contrast scans, faded text, and local lighting differences. Local contrast thus has less influence on the hash. The clip limit caps the enhancement. Without it, noise such as paper texture or compression artifacts in scanned pages would be amplified.

5.2.5 Gaussian Smoothing

A Gaussian blur with a 3×3 kernel is then applied (Listing 5.4). It suppresses high-frequency variation: isolated pixels, minor compression noise, and small antialiasing differences at character edges.

```
1 image = cv2.GaussianBlur(image, (3, 3), 0)
```

Listing 5.4: Gaussian smoothing

Without this step, the same content rendered from two different PDFs could hash differently because of a few edge pixels. The small kernel limits smoothing while retaining most

document structure in the evaluated samples. A larger kernel would remove more noise, but it could also erase thin lines or small symbols. The kernel size balances these two effects.

5.2.6 Grayscale quantization

An 8-bit grayscale image has 256 intensity levels. The implementation reduces them to 64 by uniform quantization (Listing 5.5). The divisor is $256/64 = 4$. Integer division maps each value to the lower bound of its four-value interval. Values 100 to 103 will all become 100.

```
1 QUANTIZE_LEVELS = 64
2
3 divisor = 256 // QUANTIZE_LEVELS
4 image = (image // divisor) * divisor
```

Listing 5.5: Grayscale intensity quantization

Small intensity differences from rasterization, compression, scanner noise, or antialiasing disappear when they fall into the same interval. Sixty-four levels were selected as an engineering trade-off intended to retain distinctions among text, backgrounds, diagrams, and shaded regions while reducing small intensity variations. The value is an engineering trade-off, not derived from a systematic sensitivity analysis.

5.2.7 PNG Re-encoding

OpenCV re-encodes the normalized single-channel image as PNG. These bytes are the canonical representation $b_i = \phi_v(x_i)$ and form the direct input to SHA-256.

PNG is lossless, so re-encoding adds no approximation. Under a fixed environment, it also produces stable bytes. Different encoders, settings, or library versions, however, may produce different bytes for the same pixels. Any change that can alter the canonical bytes therefore requires a new `RENDER_VERSION`. This applies to the rendering scale, the normalization parameters, the encoder, and their dependencies.

5.2.8 SHA-256 Hash Computation

SHA-256 belongs to the SHA-2 family and produces a 256-bit digest [8]. For page i , the identifier thus satisfies $h_i \in \{0, 1\}^{256}$. It is stored as the 64-character hexadecimal field `pageHashSha256` (Listing 5.6).

```
1 canonical_bytes = _normalize_image(ocr_bytes)
2 page_hash = hashlib.sha256(canonical_bytes).hexdigest()
```

Listing 5.6: SHA-256 hash computation

SHA-256 only sees a byte sequence. It knows nothing about pages, pixels, or visual similarity. All tolerance to visual variation therefore comes from the normalization steps above.

The normalized image is used only for the cache key. When OCR is required, the provider receives either the original PDF or a temporary PDF copied from it. The normalization procedure affects page identification without modifying the content supplied for text recognition.

SHA-256 suits this thesis's purpose because it is deterministic, standardized, and part of Python's standard library. Its fixed-length output can be indexed efficiently in MongoDB, and the page image need not be stored in the cache metadata.

For N_{hash} distinct canonical byte sequences, the birthday bound approximates the probability of at least one accidental collision as

$$P_{\text{coll}}(N_{\text{hash}}) \approx \frac{N_{\text{hash}}(N_{\text{hash}} - 1)}{2^{257}}.$$

Assuming uniformly distributed outputs, this probability is negligible at the evaluated scale. This justifies the collision-resistance assumption of Chapter 3.

A cryptographic collision must be distinguished from *canonicalization collapse*. Collapse occurs when two different source pages become byte-identical after normalization. SHA-256 then receives identical input, and equal digests are expected. Whether such a collapse is acceptable is exactly the semantic assumption on ϕ_v in Chapter 3.

Section 2.4 explains why explicit normalization followed by exact SHA-256 identification was preferred to perceptual image hashing.

5.2.9 Cache Lookup

After hashing, the pages are sorted by their original indices. Duplicate hashes are removed from the lookup request, keeping the first occurrence. The implementation uses `dict.fromkeys()`, which preserves insertion order (Listing 5.7).

```

1 hashes = list(
2     dict.fromkeys(page.sha256 for page in rendered_pages)
3 )

```

Listing 5.7: Page hash deduplication

The cache has two parts. MongoDB holds the searchable metadata in the collection `page_ocr_cache`. Blob Storage holds the full page-level OCR results as JSON in the container `page-ocr-cache`. The cache key $\kappa_i = (h_i, v, \theta)$ of Equation 3.9 maps to the fields `pageHashSha256`, `renderVersion`, and `ocrModel`. A unique compound index on these three fields guarantees at most one record per key.

The service issues a single query for all distinct hashes (Listing 5.8). Records must match the current `renderVersion` and `ocrModel`. They must also carry an `ocrResultAddress` that points into the page-level cache container.

```

1 cursor = collection.find(
2     {
3         "pageHashSha256": {"$in": hashes},
4         "renderVersion": render_version,
5         "ocrModel": ocr_model,
6         "ocrResultAddress": {
7             "$exists": True,
8             "$regex": f"^{PAGE_OCR_CACHE_ADDRESS_PREFIX}",
9         },
10    }
11 )

```

Listing 5.8: MongoDB query criteria

A returned record is only a candidate hit. The repository increments its `hitCount` and adds the current review identifier to `relatedReviewIds` at this point. The service then loads the referenced JSON object from Blob Storage. Only if this succeeds is the entry reused. If the object is missing or cannot be read, the candidate is rejected, and the page is treated as a cache miss. This two-stage check implements `LookupValid` of Algorithm 1.

5.2.10 Cache-Miss Pages and Temporary-PDF Construction

A page is a cache miss if its key does not resolve to a loadable result. Several pages may share the same missing key. The service scans the pages in order and keeps only the first page of each missing hash. The resulting list `missing_pages` corresponds to the representative pages $x_{Rep[q_1]}, \dots, x_{Rep[q_m]}$. If it is empty, no temporary PDF is built and no OCR request is made.

Otherwise, PyMuPDF copies these pages from the submitted PDF into a new document (Listing 5.9). The page order of this temporary PDF equals the order of `missing_pages`. This order is later used to map the OCR response back.

```
1 source_doc = fitz.open(stream=pdf_bytes, filetype="pdf")
2 miss_doc = fitz.open()
3
4 for page_index in page_indices:
5     miss_doc.insert_pdf(
6         source_doc,
7         from_page=page_index,
8         to_page=page_index,
9     )
10
11 temporary_pdf = miss_doc.tobytes()
```

Listing 5.9: Temporary PDF construction

The rendering yields a deterministic page identity. The copied pages preserve the submitted content for recognition.

An alternative design would submit every representative cache-miss page as an independent OCR request. This approach was not selected because a multi-page document could generate a burst of external API calls within a short period. Such calls introduce repeated connection, request, and serialization overhead and may also encounter provider-side rate limiting.

Instead, the representative cache-miss pages are combined into a single temporary PDF and submitted as one OCR batch. Under normal execution, this reduces a potentially large burst of page-level requests to one initial batch request per uploaded PDF that contains at least one cache miss.

The Mistral client treats HTTP 429 and several 5xx codes as transient. It retries them and may then fall back to the next configured OCR model. Batching does not rule out rate limiting, but it reduces the number of requests. If a fallback model serves the request, the new results are cached under that model's identifier.

5.2.11 Mistral OCR Processing

`OCRService.process_pdf_to_ocr()` sends the temporary PDF through `MistralAIClient`, exactly as the original workflow sends a complete PDF. The same Markdown sanitization applies.

The implementation assumes that a successful request returns one page object per input page, in input order. This matches the response-order assumption of Chapter 3. It held for all documents in the live validation (Section 5.3).

5.2.12 Mapping OCR Results Back to Original Pages

The service keeps a request-local dictionary `ocr_by_hash`. Loaded cache hits are inserted first, keyed by page hash. MongoDB operations use the full key (h_i, v, θ) . The dictionary needs only h_i , because v and θ are fixed within one request.

The OCR service returns a list `ocr_pages`. Its element at position j is assigned to the representative page at position j of `missing_pages`. The result is then stored in `ocr_by_hash` under that page's hash. All occurrences of the same hash later read this one entry. Listing 5.10 shows the mapping, the placeholder handling, and the restoration of the original order.

```

1 results = []
2 for index, page in enumerate(missing_pages):
3     page_ocr = (
4         ocr_pages[index]
5         if index < len(ocr_pages)
6         else None
7     )
8     results.append((page, page_ocr))
9
10 for page, page_ocr in results:
11     if page_ocr is None:
12         ocr_by_hash[page.sha256] = {
13             "markdown": "",
14             "page_number": page.page_index,
15             "images": [],
16         }
17         continue
18
19     ocr_by_hash[page.sha256] = page_ocr
20     pages_to_cache.append((page, page_ocr))
21
22 ocr_json_data = [
23     self._normalize_page_ocr(
24         ocr_by_hash[page.sha256],
25         page,
26     )
27     for page in rendered_pages
28 ]

```

Listing 5.10: Positional OCR-result mapping and original-order restoration

If the returned list is shorter than expected, the remaining representative pages receive an empty placeholder. Placeholders are not added to `pages_to_cache`. They never reach the persistent cache. A length mismatch is logged but not otherwise handled. The placeholder is a defensive fallback, not a correspondence check. Correct mapping still depends on the provider preserving input order.

Each successful new result is then persisted. The JSON object is written to Blob Storage first. Only then is the MongoDB record upserted with its `ocrResultAddress`. A lookup therefore never finds a record whose object is not yet stored. Writes for different pages run concurrently. Each blob write and each upsert is attempted up to three times with linear back-off. If all attempts fail, the error propagates and the whole task fails. The queue then redelivers it.

5.2.13 Result Integration and Persistence

At this stage, `ocr_by_hash` holds one OCR object for every page hash of the document. The service iterates over `rendered_pages` in original order and looks up each hash. This restores the page order of the upload.

For each occurrence, `_normalize_page_ocr()` deep-copies the selected object and sets document-specific metadata. This implements `AdaptPage`:

`page_id` `page- $k$` , where k is the one-based page position;

`page_number` the zero-based page index;

`page_sha256` the canonical page hash; and

`render_version` the value of `RENDER_VERSION`.

The copy is necessary. One object may be reused at several positions. Changing it in place would give all occurrences the same page number.

Image identifiers are made page-specific as well. Each identifier receives the prefix `page-k-`, and matching image references in the Markdown are rewritten. Without this step, two pages could contain images with the same page-local identifier.

`OCRService.extract_markdown()` then reads `page_number` and `markdown` from each object and calls `concat_pages()`. This implements Assemble. Both the original path and the reuse path use this method. Page-level reuse leaves the Markdown format unchanged.

The remaining steps are those of the original workflow. The system records the page count, image count, and OCR model, and extracts the metadata from the first two pages. `PaperStorageService.store_ocr()` then stores the review-level OCR JSON and the Markdown concurrently. When page-level reuse is active, it also stores a copy of the adapted page results for the review. These review-level artifacts are separate from the shared page cache. `PDFIngestionService.reconstruct()` later reloads the JSON for reference extraction, image processing, and review generation.

5.3 Implementation Validation

5.3.1 Automated Functional Tests

The reuse mechanism was validated with 18 automated unit tests under a fixed local configuration. The tests are grouped by the component they exercise:

- *Page identification* (`PageScreenshotRenderer`): rendering the same PDF twice yields identical page hashes; page geometry is preserved; the normalized image contains only the 64 quantized levels; and the hash equals the SHA-256 digest of the normalized bytes.
- *Cache lookup and miss handling* (`PageOCRService`): a fully cached document triggers no OCR request; mixed hits and misses are merged in original order; a record whose blob cannot be loaded is reprocessed as a miss; a page repeated within one PDF is sent to OCR only once.
- *Temporary PDF*: all distinct misses are copied from the original PDF into a single temporary PDF and sent in one request.
- *Result integration*: pages rendered out of order are sorted by index before results are combined.
- *Failure handling*: placeholders for a short OCR response are not cached; a cache write that keeps failing raises after three attempts, whereas a transient failure recovers.
- *Cache persistence* (`PageOCRCacheRepository`, `PaperStorageService`): the compound unique index exists; lookups and upserts use the full key; concurrent insertion of the same key is tolerated; page results are stored under a hash-derived blob address.

All 18 test cases passed. Together, they confirm that distinct misses are copied into one PDF in their original relative order and submitted in one OCR request. They also confirm that cached and newly produced results are restored to their original positions and that repeated page content is adapted separately for each occurrence.

These tests validate control flow and data mappings under the tested configuration. They do not exercise the real OCR provider.

5.3.2 Limited Live Integration Validation

Two live integration tests exercise the page-level reuse path against the running API service, MongoDB, Azure Blob Storage, and the Mistral OCR API.

The first test checks page selection and batching. Before uploading a PDF, it renders the pages locally and counts the distinct hashes that are not yet cached. It then uploads the PDF through the API and runs the page-level OCR path on the stored file. The test asserts that:

- at most one OCR request is made, and exactly one if any page is missing;
- the temporary PDF contains exactly as many pages as there are distinct missing hashes;
- no temporary PDF is created if every page is cached;
- the output contains one result per source page and non-empty Markdown; and
- every cache record points to `page-ocr-cache/<hash>.json` and lists the review identifier.

The second test checks reuse across submissions. It uploads the same paper twice through the API. After the first upload, the cache record carries the expected blob address, render version, and OCR model. The second upload yields the same page hash and identical Markdown, and the record's `hitCount` increases.

In addition, a small set of research papers was processed manually on the same infrastructure. Each page received a non-empty OCR result. The response order matched the page order of the temporary PDF. Spot checks confirmed that the Markdown stayed aligned with its source pages. When the papers were submitted again, a fully cached submission triggered no OCR request and produced the same document-level Markdown.

These observations show that page selection, batch OCR, cache population, cache retrieval, and reconstruction work together under the tested conditions. However, the sample was small. The validation should therefore not be interpreted as a large-scale assessment of OCR equivalence, cross-environment hash stability, adversarial cache integrity, performance overhead, or document-layout diversity.

6 Evaluation

This chapter evaluates the page-level OCR reuse mechanism along two dimensions. The first is how much page-level reuse is observable in the production workload and how widely it is distributed across reviewed PDFs. The second is how many external OCR page operations are avoided and what is known about the overhead the cache introduces.

The feature was enabled for a non-public subset of production traffic, so the results describe that cohort rather than platform-wide traffic. The evaluation is limited to reuse, avoided work, and storage footprint. Because per-stage timers were not in place, end-to-end latency, throughput and total operating cost were not measured, and no claim about them is made in this chapter.

6.1 Data and Measurement Method

The production analysis uses an aggregate snapshot of the page OCR cache collection. The observation window is approximately 38 days.

The workload quantities of Section 3.1.2 are reconstructed in aggregate from the cache state $\mathcal{C}_T$ at the snapshot, where T denotes the last submission before the snapshot. Under the normal processing path, submission t creates one entry for each of its m_t distinct missing keys and increments *hitCount* once for each of its $d_t - m_t$ distinct keys found in the cache.

The number of entries and the sum of all recorded hits therefore satisfy

$$N_{\text{entry}} = |\text{dom}(\mathcal{C}_T)| = \sum_{t=1}^T m_t, \quad N_{\text{hit}} = \sum_{e \in \text{dom}(\mathcal{C}_T)} \text{hitCount}(e) = \sum_{t=1}^T (d_t - m_t),$$

and their sum $N_{\text{entry}} + N_{\text{hit}} = \sum_t d_t$ is the number of observable distinct-key lookups. The aggregate hit rate is

$$\bar{\rho}^{\text{hit}} = \frac{\sum_t (d_t - m_t)}{\sum_t d_t} = \frac{N_{\text{hit}}}{N_{\text{entry}} + N_{\text{hit}}}.$$

As noted in Section 3.1.2, this is a distinct-key rate. It does not count repeated occurrences of a cached page within one document, and it may therefore underestimate the occurrence-based rate ρ_t^{cache} aggregated over submissions. The page count n_t of each submission is stored in the review metadata but is not included in the aggregate snapshot. The audit did, however, compare it with the number of cache entries associated with each review. Because a review is recorded at most once per entry, this number equals d_t . Of the 5,885 reviews associated with cache entries, 5,810 could be matched to a review record; for 75 no record was found. The two counts agreed for 5,806 of the 5,810 matched reviews, so within-document duplicates ($n_t > d_t$) are rare in the observed workload and the difference between the two rates is small. The denominator of $\bar{\rho}^{\text{hit}}$ also excludes requests that leave no trace in the retained cache state. $\bar{\rho}^{\text{hit}}$ is therefore an observable hit rate, not a complete measurement of production traffic.

Review-level coverage is reconstructed independently from the `relatedReviewIds` array of each entry. The first element is treated as the populating occurrence and later elements as reuse occurrences. These occurrences are grouped by review and classified as no hit, partial hit, or full hit.

Three ratios with different denominators are reported below: the *page hit rate* (hits per distinct-key lookup), the *entry reuse rate* (entries with at least one hit per entry) and the *review benefit rate* (reviews with at least one reused page per associated review).

6.2 Workload Reuse

6.2.1 Overall Page Reuse

The retained cache contained $N_{\text{entry}} = 69,462$ entries and recorded $N_{\text{hit}} = 24,035$ hits. This gives $\sum_t d_t = 93,497$ observable distinct-key lookups and a page hit rate of $\bar{\rho}^{\text{hit}} = 25.71\%$, shown in Figure 6.1. Assuming that each recorded hit replaced a new OCR operation, the hits correspond to 24,035 avoided external OCR page operations.

Cache populating requests 69,462 (74.29%)	Recorded hits 24,035 (25.71%)
---	---

Figure 6.1: Composition of 93,497 observable distinct-key lookups reconstructed from the retained cache state. The observable page hit rate is $\bar{\rho}^{\text{hit}} = 25.71\%$.

Of the 69,462 entries, 13,881 (19.98%) had been reused at least once by the snapshot date. This entry reuse rate is lower than the page hit rate because it counts cached pages rather than lookups: a single entry can serve many hits, and the most frequently reused entry recorded 25. Grouped by creation week, the five complete weekly cohorts had reconstructed hit rates between 20.97% and 30.63%. Reuse was present in every cohort during the observation period.

6.2.2 Coverage at PDF-Review Level

Of 5,885 associated reviews, 1,204 (20.46%) were classified as fully cached, 594 (10.09%) as partially cached and 4,087 (69.45%) as having no cache benefit. In total, 1,798 reviews (30.55%) benefited from at least one reused page. Figure 6.2 shows the distribution over all associated reviews and the composition of the benefiting subset.

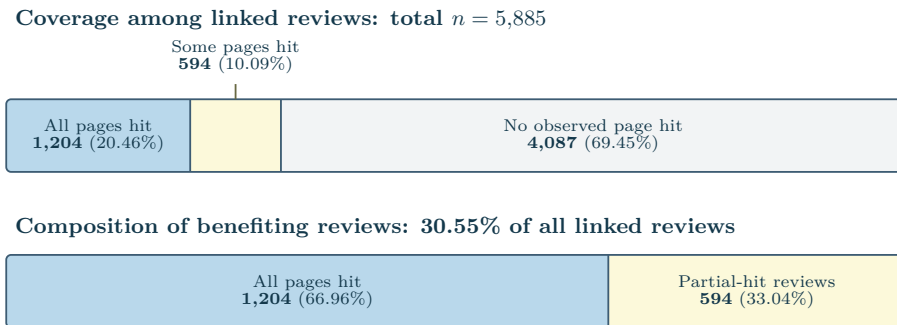
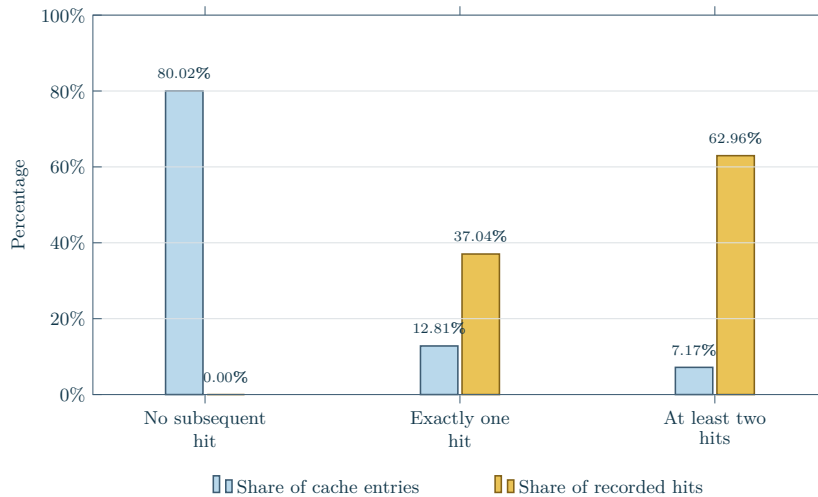


Figure 6.2: Cache benefit across 5,885 associated reviews. Partially cached reviews account for 33.04% of the 1,798 benefiting reviews. Categories are reconstructed from cache-to-review associations and are therefore approximate.

Partially cached reviews make up 33.04% of the benefiting reviews. They contain a mix of reusable and non-reusable pages. The result shows that partial reuse occurs in the observed workload, which motivates the choice of page granularity. It does not quantify the additional savings of page-level over whole-file caching, which would require evaluating both schemes on the same workload.

6.2.3 Concentration of Reuse

Reuse was unevenly distributed across entries, as shown in Figure 6.3. 55,581 entries (80.02%) had no recorded hit. A further 8,902 entries (12.81%) had exactly one hit and produced 37.04% of all hits. The 4,979 entries with at least two hits represented only 7.17% of the stored entries but produced 15,133 hits, or 62.96% of the total. The highest hit count of any single entry was 25.



Note: The two series use different denominators: 69,462 cache entries and 24,035 recorded hits.

Figure 6.3: Distribution of retained entries and recorded hits by reuse frequency. Entries with at least two hits account for 7.17% of retained entries and 62.96% of recorded hits.

The aggregate hit rate does not come from uniform reuse across the cache. Most of the observed benefit comes from a comparatively small set of repeatedly requested pages. This pattern is consistent with repeated submissions, revisions, and shared or duplicated material.

The zero-hit share should not be read as a set of entries that will never be reused. Entries created shortly before the snapshot have had less opportunity to be hit. Reconstructed hit rates by entry age reflect this. Entries aged 0–1 days reach 20.52%, those aged 2–7 days 21.45%, and those aged 8–30 days 27.43%. A retention or eviction policy would have to account for entry age, and the present window is too short to calibrate one.

6.2.4 Temporal Locality of Reuse

For the 13,881 reused entries, the audit measures the span between `created` and `modified`. The median span was 18.13 minutes. Of the reused entries, 65.95% had a span of at most one hour, 86.08% at most 24 hours, 96.46% at most seven days, 98.03% at most 14 days and 99.65% at most 30 days. Figure 6.4 shows the cumulative distribution on a logarithmic time axis.

The distribution shows strong short-term temporal locality. A typical reused entry is hit again soon after its creation, which is consistent with the same document being resubmitted several times within a short period. This suggests that retention experiments should pay particular attention to the first day and the first week after an entry is created.

The measure has two limitations. First, it is the span from creation to the latest recorded modification, not the delay to the first hit or a timestamp for each individual hit. Second,

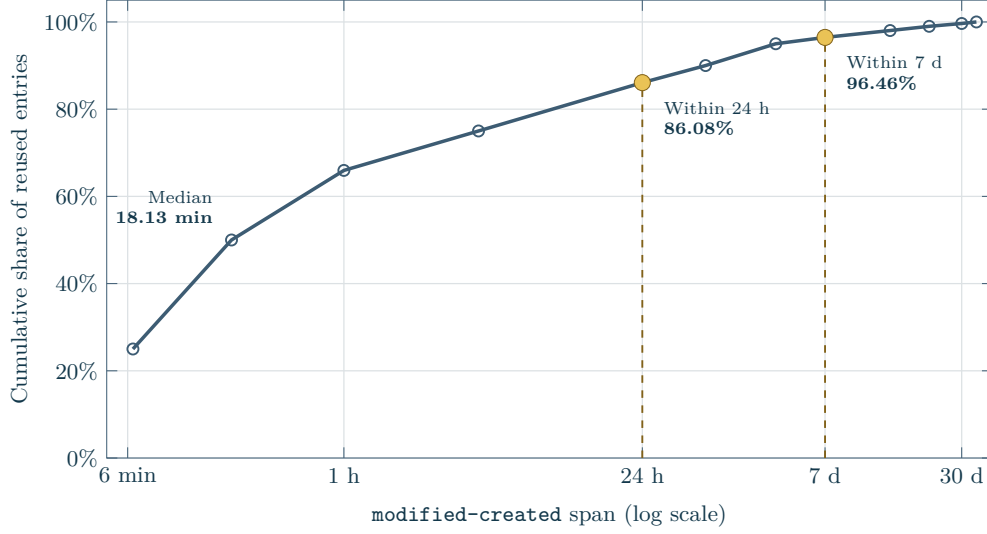


Figure 6.4: Cumulative distribution of the time between creation and last recorded reuse (modified – created) for the 13,881 reused entries. The span is not the time to first hit, and the distribution is weighted by entry, not by hit.

the percentages are weighted by entry rather than by hit. The result therefore does not imply that a seven-day time-to-live would retain 96.46% of future hits.

6.3 Avoided Cost and Overhead

6.3.1 Avoided OCR Work

By Equation 3.17, submission t avoids $n_t - m_t$ OCR pages compared with full processing. This quantity splits into two parts:

$$n_t - m_t = (n_t - d_t) + (d_t - m_t).$$

The first part comes from within-document duplicates, which are collapsed before lookup. The second part comes from distinct keys found in the persistent cache. The snapshot records the second part, summed over submissions as N_{hit} . The first part is not included in the snapshot, so the total number of avoided OCR pages is bounded from below by

$$\sum_{t=1}^T (n_t - m_t) \geq \sum_{t=1}^T (d_t - m_t) = N_{\text{hit}} = 24,035.$$

Since $n_t = d_t$ held for 5,806 of the 5,810 matched reviews (Section 6.1), the first part is close to zero and this bound is nearly tight. Relative to processing every observable distinct key, the cache removed $\rho^{\text{hit}} = 25.71\%$ of the external OCR page operations.

With c_f the per-page OCR cost of Equation 3.18, the avoided OCR cost over the observation window is at least $N_{\text{hit}} c_f$. This corresponds to the first term $(n_t - m_t)c_f$ of Equation 3.21, summed over submissions. It is a gross quantity. It states how much OCR work was avoided, not how much the system as a whole saved, because the reuse overhead O_t has not yet been subtracted.

6.3.2 Break-Even Condition and Storage Footprint

Summing Equation 3.22 over the submissions in the observation window, the cache yields a net benefit in the chosen cost dimension if

$$\sum_{t=1}^T O_t < c_f \sum_{t=1}^T (n_t - m_t).$$

Because the right-hand side is at least $N_{\text{hit}} c_f$, the observed data give the sufficient condition

$$\sum_{t=1}^T O_t < N_{\text{hit}} c_f.$$

Dividing both sides by the number of observable distinct-key lookups $\sum_t d_t$ yields

$$\frac{\sum_t O_t}{\sum_t d_t} < \bar{\rho}^{\text{hit}} c_f \approx 0.26 c_f.$$

In words, the cache is beneficial if its average overhead per observable lookup stays below about one quarter of the cost c_f of processing one page with OCR. This is the aggregate, observable counterpart of Equation 3.23. It is expressed relative to c_f and therefore does not depend on the absolute value of c_f . Because the condition uses a lower bound on the avoided pages, it is conservative. Since within-document duplicates are rare, however, the actual margin is only slightly larger.

The audit does not measure O_t , and so it cannot establish whether the implementation satisfies the condition. The terms of O_t correspond to the following steps of the processing path in Chapter 5:

- $n_t c_{\text{id}}$, **rendering, canonicalization and hashing**. This work is performed for every page.
- $C_{\text{lookup}}(d_t)$, **lookup and result retrieval**. Every submission incurs one batched index query for its d_t distinct keys, and each of the 24,035 recorded hits required loading the stored OCR result from object storage.
- $C_{\text{pack}}(m_t)$, **temporary PDF**. When a submission has misses, a temporary PDF containing the m_t representative pages is built before OCR. It occurs in every submission with at least one miss, including the 594 partial-hit reviews.
- $C_{\text{store}}(m_t)$, **population**. Each of the 69,462 entries required an object write and an index upsert, with retries on transient failures.
- $C_{\text{merge}}(n_t)$, **result mapping**. Cached and new results are mapped back to their original positions before assembly.

Whether the cache reduces processing time overall likewise depends on the balance between $\bar{\rho}^{\text{hit}}$ and the per-stage times, which were not recorded.

For the cache storage of Equation 3.24, the snapshot gives $S_{\text{data}} = 10.97$ MiB of allocated document storage and $S_{\text{index}} = 18.68$ MiB of index storage, so that $S_c = 29.64$ MiB. Indexes account for 63.00% of this. The OCR result objects are stored externally in a dedicated object-storage container and are not included in this figure. The total storage cost of the cache is therefore higher than the MongoDB footprint alone.

6.4 Summary

In summary, the cache avoided at least 24,035 OCR page operations, a quarter of the observable distinct-key lookups, and this benefit reached 30.55% of the associated reviews. It is concentrated in a small set of repeatedly and quickly reused pages. Against this, every page incurs identification cost, every submission a lookup, every miss packing and storing, and all entries consume storage. The overhead O_t , the latency effect and the size of the externally stored OCR results were not measured, so whether the net effect is positive cannot be established from the present data. The 38-day window is also too short to observe long-term reuse or to calibrate a retention policy.

7 Discussion and Limitations

Chapter 6 reported what the cache did in production. This chapter interprets those results, sets out the assumptions that the evaluation could not verify, and outlines the work needed before the mechanism can be judged in full.

7.1 Discussion

What page granularity enables. Page-level reuse does not change the external interface of the pipeline. Registration, Markdown synthesis, metadata extraction and review generation are the same as before; only the set of pages sent to OCR changes. The observed avoided OCR work comes from reuse of one intermediate result, not from changes elsewhere in the workflow.

The review-level results show why granularity matters. The 594 partially cached reviews are revisions or recombinations in which some pages had changed. A whole-file cache would have missed all of them, because any edit changes the file digest. Even the 1,204 fully cached reviews are not necessarily byte-identical resubmissions: a page-level full hit can be assembled from pages that first appeared in different earlier documents. Whole-file caching would capture only a subset of these reviews.

The design separates two roles that the original workflow combined. The page is the unit of identification, reuse, and cost accounting, whereas the PDF remains the unit of transport. Because all misses are packed into one temporary PDF, finer-grained reuse does not produce one external request per page.

Precision over recall. The mechanism accepts only exact matches of canonical bytes. Near-duplicate or perceptual matching (Section 2.4) could raise the hit rate, but in academic papers a small visual change often carries meaning: a changed digit in a table or a flipped sign in an equation. Reusing the earlier OCR output in such a case would silently put wrong content into the review input, and the error might not be detected downstream. A missed hit costs one extra OCR page, while a false hit corrupts a result. The asymmetry justifies choosing precision.

Exact hashing does not make every match safe. The tolerance of the mechanism lies entirely in the canonicalization ϕ_v . Grayscale conversion, contrast normalization, blurring and quantization are all lossy and can make two different pages byte-identical. The main semantic risk is therefore canonicalization collapse. The parameters were chosen as engineering trade-offs. No false match was observed in the limited validation, but the sample was too small to estimate the false-match rate.

What the reuse pattern implies for operation.

For most reused entries, the observed span between creation and the last recorded update was short: 65.95% had a span of at most one hour and 86.08% at most 24 hours (Section 6.2.4). This pattern is consistent with repeated submissions occurring within a short working period, although the snapshot does not record the time of each individual hit. The observed cache therefore resembles a short-lived working set with a long tail.

Two design implications follow. First, retention can probably be much more selective than the current keep-everything policy, but the first day and first week after creation deserve the most care, and entry age must be taken into account. Second, the fixed overhead

matters. Every page is rendered, normalized and hashed, including for the 69.45% of associated reviews that had no recorded persistent-cache hit. By the break-even condition of Section 6.3.2, the average overhead per lookup must stay below about one quarter of the per-page OCR cost. A cheap document-level check, such as a digest of the uploaded bytes consulted before page rendering, could serve byte-identical resubmissions directly, thereby avoiding page rendering, normalization, and lookup in that case. The present data do not show how many of the fully cached reviews were byte-identical resubmissions.

7.2 Limitations

Snapshot-based reconstruction. All workload figures were reconstructed from a single snapshot of the retained cache state (Section 6.1), rather than from a request log. Requests that failed before populating the cache and entries removed before the snapshot cannot be reconstructed. Moreover, the temporal analysis measures only the span from creation to the latest recorded update because individual hit timestamps were not retained. Consequently, time to first reuse and the distribution of per-hit delays cannot be determined.

Scope of the workload. The data cover a non-public cohort of one service over about 38 days. They do not establish a stable long-term hit rate, which may vary with submission deadlines, user growth and product changes. Entries created late in the window had less opportunity to be reused, so the 80.02% zero-hit share overstates how many entries will never be reused. Pipelines for invoices, books or scanned archives may show very different page stability.

Correctness assumptions that were not verified at scale. Several correctness assumptions stated in Chapter 3 were checked only in the small validation of Section 5.3.

- *Semantic adequacy of ϕ_v :* canonicalization collapse has not been measured.
- *Response order:* new results are mapped back by position (Section 5.2.12). A shorter response is logged and padded with placeholders, but a reordered or merged page would not be detected and would be cached under the wrong key.
- *Page independence:* reuse assumes that the OCR result of a page does not depend on the pages submitted with it. A page recognized inside a temporary PDF may, in principle, differ from the same page recognized in its original document.

Key stability has a similar weakness on the client side. `RENDER_VERSION` is maintained manually. A pipeline change that alters the canonical bytes can cause avoidable misses against existing entries. More seriously, a change that affects result compatibility without a corresponding version update can allow results from incompatible processing configurations to share the same logical key space.

Unmeasured costs. The evaluation establishes a lower bound on avoided OCR pages, not a net saving. The overhead O_t , end-to-end latency and the size of the externally stored OCR results were not measured (Section 6.3.2), so it remains open whether the break-even condition holds in practice.

Data governance. The cache is shared across users and currently has no eviction. Its entries outlive the reviews that created them and record their review identifiers. Deletion requests, retention obligations and access control on a shared cache were outside the scope of this thesis. They must be addressed before the mechanism is applied to workloads with stronger confidentiality or deletion requirements.

7.3 Future Work and Broader Applicability

Most of the limitations above can be addressed with better telemetry. An append-only event record per page occurrence, containing the request, page position, cache key, hit or miss outcome, timestamp and OCR submission, would replace the reconstruction with direct measurement and would give exact reuse delays.

Per-stage timers for rendering, lookup, packing, OCR and storage would allow the break-even condition to be tested directly. On this basis, retention and eviction policies could be compared by the OCR work they preserve against their storage and lookup costs.

The unverified assumptions should be evaluated through systematic differential and adversarial tests. Pages containing controlled changes, such as altered digits, signs, colors, or layout elements, could be used to estimate whether canonicalization maps semantically different pages to identical bytes. Synthetic batches with uniquely identifiable pages could test response count, order, schema conformance, and page-to-result correspondence. Finally, the OCR output for the same page could be compared when the page is processed alone, in its original document, and in temporary PDFs containing different neighboring pages.

Beyond OCR, the underlying pattern is to split an expensive external computation into stable, independently processable units, identify each unit by a versioned content key, and reuse earlier outputs before calling the service. Page-level layout analysis, table and formula recognition, translation, and information extraction are candidates.

The pattern applies only if three conditions hold: the units must be processed largely independently of their context, their outputs must be mapped reliably back to their original positions, and the processing configuration relevant to result compatibility must be represented in the key. The contribution of this thesis is not a universal page cache, but a design pattern for pipelines in which expensive computation can be partitioned into stable units with versioned identities and position-preserving result assembly.

Bibliography

- [1] Shakked Noy and Whitney Zhang. “Experimental evidence on the productivity effects of generative artificial intelligence”. In: *Science* 381.6654 (2023), pp. 187–192. DOI: 10.1126/science.adh2586.
- [2] Erik Brynjolfsson, Danielle Li, and Lindsey Raymond. “Generative AI at Work”. In: *Quarterly Journal of Economics* 140.2 (2025), pp. 889–942. DOI: 10.1093/qje/qjae044.
- [3] Emma Strubell, Ananya Ganesh, and Andrew McCallum. “Energy and Policy Considerations for Deep Learning in NLP”. In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Ed. by Anna Korhonen, David Traum, and Lluís Màrquez. Florence, Italy: Association for Computational Linguistics, July 2019, pp. 3645–3650. DOI: 10.18653/v1/P19-1355.
- [4] Nikolaos Livathinos et al. “Robust PDF Document Conversion Using Recurrent Neural Networks”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 35.17 (2021), pp. 15137–15145. DOI: 10.1609/aaai.v35i17.17777.
- [5] Lele Cao, Lei You, and R&D Team. “CSPaper Review: Fast, Rubric-Faithful Conference Feedback”. In: *PROCEEDINGS OF THE 18TH INTERNATIONAL NATURAL LANGUAGE GENERATION CONFERENCE: SYSTEM DEMONSTRATIONS*. Ed. by L Flek et al. Association for Computational Linguistics, 2025, pp. 3–7. URL: <https://aclanthology.org/2025.inlg-demos.2/>.
- [6] Umut A. Acar, Guy E. Blelloch, and Robert Harper. “Selective memoization”. In: *ACM SIGPLAN Notices* 38.1 (2003), pp. 14–25. DOI: 10.1145/640128.604133.
- [7] Sean Quinlan and Sean Dorward. “Venti: A New Approach to Archival Data Storage”. In: *Conference on File and Storage Technologies (FAST 02)*. Monterey, CA: USENIX Association, Jan. 2002, pp. 89–101. URL: <https://www.usenix.org/conference/fast-02/venti-new-approach-archival-data-storage>.
- [8] National Institute of Standards and Technology. *Secure Hash Standard (SHS)*. Federal Information Processing Standards Publication (FIPS PUB) 180-4. U.S. Department of Commerce, Aug. 2015. DOI: 10.6028/NIST.FIPS.180-4. URL: <https://csrc.nist.gov/pubs/fips/180-4/upd1/final>.
- [9] D MICHIE. “MEMO FUNCTIONS AND MACHINE LEARNING”. In: *NATURE* 218.5136 (1968), pp. 19–22. DOI: 10.1038/218019a0.
- [10] W. Pugh and T. Teitelbaum. “Incremental computation via function caching”. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’89. Association for Computing Machinery, 1989, pp. 315–328. ISBN: 0897912942. DOI: 10.1145/75277.75305.
- [11] Umut A. Acar, Guy E. Blelloch, and Robert Harper. “Adaptive functional programming”. In: *ACM Transactions on Programming Languages and Systems* 28.6 (Nov. 2006), pp. 990–1034. ISSN: 0164-0925. DOI: 10.1145/1186632.1186634.
- [12] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. “Build systems à la carte”. In: *Proc. ACM Program. Lang.* 2.ICFP (July 2018). DOI: 10.1145/3236774.
- [13] E Dolstra, M de Jonge, and E Visser. “Nix: A safe and policy-free system for software deployment”. In: *Proceedings of the 18th Large Installation System Administration Conference (LISA ’04)*. USENIX Association, 2004, pp. 79–92.
- [14] Pradeep Kumar Gunda et al. “NeCTAR: Automatic management of data and computation in datacenters”. In: *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. USENIX Association, 2010, pp. 75–88. URL:

- <https://www.usenix.org/conference/osdi10/nectar-automatic-management-data-and-computation-datacenters>.
- [15] Iman Elghandour and Ashraf Abounaga. “ReStore: Reusing results of MapReduce jobs”. In: *Proceedings of the VLDB Endowment* 5.6 (2012), pp. 586–597. DOI: 10.14778/2168651.2168659.
 - [16] Lucian Popa et al. “DryadInc: reusing work in large-scale computations”. In: *Workshop on Hot Topics in Cloud Computing (HotCloud 09)*. USENIX Association, June 2009. URL: <https://www.usenix.org/conference/hotcloud-09/dryadinc-reusing-work-large-scale-computations>.
 - [17] Doris Xin et al. “HELIX: Holistic optimization for accelerating iterative machine learning”. In: *Proceedings of the VLDB Endowment* 12.4 (2018), pp. 446–460. DOI: 10.14778/3297753.3297763.
 - [18] Behrouz Derakhshan et al. “Materialization and Reuse Optimizations for Production Data Science Pipelines”. In: *Proceedings of the 2022 International Conference on Management of Data*. SIGMOD ’22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 1962–1976. ISBN: 9781450392495. DOI: 10.1145/3514221.3526186.
 - [19] Dan Graur et al. “Cachew: Machine Learning Input Data Processing as a Service”. In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, July 2022, pp. 689–706. ISBN: 978-1-939133-29-8. URL: <https://www.usenix.org/conference/atc22/presentation/graur>.
 - [20] In Gim et al. “Prompt Cache: Modular Attention Reuse for Low-Latency Inference”. In: *Proceedings of Machine Learning and Systems*. Ed. by P. Gibbons, G. Pekhimenko, and C. De Sa. Vol. 6. 2024, pp. 325–338. URL: https://proceedings.mlsys.org/paper_files/paper/2024/file/a66caa1703fe34705a4368c3014c1966-Paper-Conference.pdf.
 - [21] Bill Bolosky et al. “Single Instance Storage in Windows 2000”. In: *4th USENIX Windows Systems Symposium*. Seattle, WA: USENIX Association, Aug. 2000. URL: <https://www.usenix.org/conference/4th-usenix-windows-systems-symposium/single-instance-storage-windows-2000>.
 - [22] Sean Rhea, Russ Cox, and Alex Pesterev. “Fast, inexpensive content-addressed storage in foundation”. In: *2008 USENIX Annual Technical Conference (USENIX ATC 08)*. USENIX Association, 2008, pp. 143–156. URL: <https://www.usenix.org/conference/2008-usenix-annual-technical-conference/fast-inexpensive-content-addressed-storage>.
 - [23] Andrew Tridgell. “Efficient Algorithms for Sorting and Synchronization”. PhD thesis. The Australian National University, Feb. 1999.
 - [24] Athicha Muthitacharoen, Benjie Chen, and David Mazières. “A low-bandwidth network file system”. In: *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP ’01)*. 2001, pp. 174–187. DOI: 10.1145/502059.502052.
 - [25] Purushottam Kulkarni et al. “Redundancy elimination within large collections of files”. In: *2004 USENIX Annual Technical Conference (USENIX ATC 04)*. USENIX Association, 2004, pp. 59–72. URL: <https://www.usenix.org/conference/2004-usenix-annual-technical-conference/redundancy-elimination-within-large-collections>.
 - [26] Calicrates Policroniades and Ian Pratt. “Alternatives for detecting redundancy in storage systems data”. In: *2004 USENIX Annual Technical Conference (USENIX ATC 04)*. USENIX Association, 2004, pp. 73–86. URL: <https://www.usenix.org/conference/2004-usenix-annual-technical-conference/alternatives-detecting-redundancy-storage-systems>.

- [27] Dutch T. Meyer and William J. Bolosky. "A study of practical deduplication". In: *ACM Transactions on Storage* 7.4 (Feb. 2012). ISSN: 1553-3077. DOI: 10.1145/2078861.2078864.
- [28] Benjamin Zhu, Kai Li, and Hugo Patterson. "Avoiding the disk bottleneck in the data domain deduplication file system". In: *6th USENIX Conference on File and Storage Technologies (FAST 08)*. USENIX Association, 2008, pp. 269–282. URL: <https://www.usenix.org/conference/fast-08/avoiding-disk-bottleneck-data-domain-deduplication-file-system>.
- [29] Ahmed El-Shimi et al. "Primary Data Deduplication—Large Scale Study and System Design". In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, 2012, pp. 285–296. ISBN: 978-1-931971-93-5. URL: <https://www.usenix.org/conference/atc12/technical-sessions/presentation/el-shimi>.
- [30] Wen Xia et al. "FastCDC: A Fast and Efficient Content-Defined Chunking Approach for Data Deduplication". In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, 2016, pp. 101–114. URL: <https://www.usenix.org/conference/atc16/technical-sessions/presentation/xia>.
- [31] Andrei Z. Broder. "Some applications of Rabin's fingerprinting method". In: *Sequences II*. Ed. by Renato Capocelli, Alfredo De Santis, and Ugo Vaccaro. New York, NY: Springer New York, 1993, pp. 143–152. ISBN: 978-1-4613-9323-8. DOI: 10.1007/978-1-4613-9323-8_11.
- [32] Phillip Rogaway and Thomas Shrimpton. "Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance". In: *Fast Software Encryption*. Ed. by Bimal Roy and Willi Meier. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 371–388. ISBN: 978-3-540-25937-4. DOI: 10.1007/978-3-540-25937-4_24.
- [33] Ramarathnam Venkatesan et al. "Robust image hashing". In: *Proceedings of the 2000 IEEE International Conference on Image Processing (ICIP 2000)*, Vol. 3. 2000, pp. 664–666. DOI: 10.1109/ICIP.2000.899541.
- [34] V. Monga and B.L. Evans. "Perceptual Image Hashing Via Feature Points: Performance Evaluation and Tradeoffs". In: *IEEE Transactions on Image Processing* 15.11 (2006), pp. 3452–3465. DOI: 10.1109/TIP.2006.881948.
- [35] Qingying Hao et al. "It's not what it looks like: Manipulating perceptual hashing based applications". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*. 2021, pp. 69–85. DOI: 10.1145/3460120.3484559.
- [36] Lukas Struppek et al. "Learning to break deep perceptual hashing: The use case NeuralHash". In: *Proceedings of the 2022 5th ACM Conference on Fairness, Accountability, and Transparency (FAccT 2022)*. 2022, pp. 58–69. DOI: 10.1145/3531146.3533073.
- [37] Chris Lamb and Stefano Zacchiroli. "Reproducible builds: Increasing the integrity of software supply chains". In: *IEEE Software* 39.2 (2022), pp. 62–70. DOI: 10.1109/MS.2021.3073045.
- [38] Venkata N. Padmanabhan and Lili Qiu. "The content and access dynamics of a busy Web site: findings and implications". In: *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '00. New York, NY, USA: Association for Computing Machinery, 2000, pp. 111–123. ISBN: 1581132239. DOI: 10.1145/347059.347413.
- [39] Nimrod Megiddo and Dharmendra S. Modha. "ARC: A self-tuning, low overhead replacement cache". In: *Proceedings of the 2nd USENIX Conference on File and*

- Storage Technologies (FAST '03)*. 2003, pp. 115–130. URL: <https://www.usenix.org/conference/fast-03/arc-self-tuning-low-overhead-replacement-cache>.
- [40] Andrei Z. Broder. “On the resemblance and containment of documents”. In: *Proceedings of the International Conference on Compression and Complexity of Sequences*. IEEE, 1997, pp. 21–29. DOI: 10.1109/SEQUEN.1997.666900.
 - [41] Kyle Williams and C. Lee Giles. “Near duplicate detection in an academic digital library”. In: *Proceedings of the 2013 ACM Symposium on Document Engineering*. DocEng '13. Florence, Italy: Association for Computing Machinery, 2013, pp. 91–94. ISBN: 9781450317894. DOI: 10.1145/2494266.2494312.
 - [42] Lukas Blecher et al. “Nougat: Neural Optical Understanding for Academic Documents”. In: *International Conference on Learning Representations*. Ed. by B. Kim et al. 2024. URL: https://proceedings.iclr.cc/paper_files/paper/2024/file/a39a9aceda771cded859ae7560530e09Paper-Conference.pdf.
 - [43] Kyle Lo et al. “PaperMage: A Unified Toolkit for Processing, Representing, and Manipulating Visually-Rich Scientific Documents”. In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, 2023, pp. 495–507. DOI: 10.18653/v1/2023.emnlp-demo.45.
 - [44] Linke Ouyang et al. “OmniDocBench: Benchmarking Diverse PDF Document Parsing with Comprehensive Annotations”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2025, pp. 24838–24848. URL: https://openaccess.thecvf.com/content/CVPR2025/html/Ouyang_OmniDocBench_Benchmarking_Diverse_PDF_Document_Parsing_with_Comprehensive_Annotations_CVPR_2025_paper.html.

A Appendix

A.1 SHA-256

SHA-256 is used in this thesis as a deterministic content identifier, not for confidentiality. The National Institute of Standards and Technology specifies SHA-256 as part of the Secure Hash Standard and describes message digests as a means of detecting whether input data has changed [8].

A cryptographic hash function maps an input of arbitrary length to a fixed-length output known as a hash value or message digest.

Before SHA-256 processing begins, an input of length $\ell < 2^{64}$ bits is padded by appending a single one bit, followed by the smallest number of zero bits such that the resulting length is congruent to 448 modulo 512. A 64-bit big-endian representation of the original message length ℓ is then appended. The padded message is divided into 512-bit blocks.

SHA-256 maintains eight 32-bit working variables, denoted by a, b, c, d, e, f, g , and h . At the beginning of each 512-bit message block, these variables are initialized using the current intermediate hash state.

For the first block, the initial state consists of eight predefined constants specified in the SHA-256 standard, obtained from the first 32 bits of the fractional parts of the square roots of the first eight prime numbers. For subsequent blocks, the state produced after processing the preceding block is used.

Each message block is processed through 64 rounds, indexed by $t = 0, 1, \dots, 63$. During round t , the algorithm calculates two temporary 32-bit values:

$$T_1 = h + \Sigma_1(e) + \text{Ch}(e, f, g) + K_t + W_t \pmod{2^{32}}$$

$$T_2 = \Sigma_0(a) + \text{Maj}(a, b, c) \pmod{2^{32}}$$

Here, W_t is the t -th word of the message schedule and therefore introduces information derived from the current message block.

K_t is the predefined constant assigned to round t . SHA-256 uses 64 different round constants, which are derived from the fractional parts of the cube roots of the first 64 prime numbers [8].

The choice function is defined as

$$\text{Ch}(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$$

For every bit position, Ch selects the corresponding bit from y when the bit in x is 1; otherwise, it selects the corresponding bit from z .

In the round computation, e therefore controls whether each output bit is taken from f or g .

The majority function is defined as

$$\text{Maj}(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

For each bit position, this function returns the value shared by at least two of its three inputs. It combines the current values of a , b , and c and introduces a nonlinear dependency between them.

The two uppercase sigma functions mix the bits within the working variables through several circular right rotations:

$$\Sigma_0(x) = \text{ROTR}^2(x) \oplus \text{ROTR}^{13}(x) \oplus \text{ROTR}^{22}(x)$$

$$\Sigma_1(x) = \text{ROTR}^6(x) \oplus \text{ROTR}^{11}(x) \oplus \text{ROTR}^{25}(x)$$

A right rotation moves every bit to the right while returning the bits that leave the right-hand side to the left-hand side. Therefore, no bits are discarded. For a 32-bit word x , rotation by n positions can be expressed as

$$\text{ROTR}^n(x) = (x \gg n) \vee ((x \ll (32 - n)) \wedge (2^{32} - 1)).$$

The use of several rotation distances distributes the influence of each input bit across different positions. The XOR operations then combine the rotated values, causing changes in one part of a working variable to affect several parts of the round output.

After T_1 and T_2 have been calculated, the eight working variables are updated as follows:

$$\begin{aligned} h &\leftarrow g \\ g &\leftarrow f \\ f &\leftarrow e \\ e &\leftarrow d + T_1 \pmod{2^{32}} \\ d &\leftarrow c \\ c &\leftarrow b \\ b &\leftarrow a \\ a &\leftarrow T_1 + T_2 \pmod{2^{32}} \end{aligned}$$

Most variables receive the value of the preceding variable, while the new values of a and e incorporate the round results. Consequently, information derived from W_t , the round constant K_t , and the current working state is repeatedly propagated through all eight variables.

Since each round depends on the state produced by the previous round, a small difference in the input can influence an increasing number of bits as processing continues.

All additions are performed modulo 2^{32} . This means that only the lowest 32 bits of each result are retained. If an addition produces a value equal to or greater than 2^{32} , the overflow is discarded. For example,

$$(2^{32} - 1) + 1 \equiv 0 \pmod{2^{32}}$$

Modulo addition keeps every working variable within the range of a 32-bit unsigned integer while introducing carry propagation that complements the bitwise XOR, rotation, and selection operations.

Right shifts are also used when constructing the message schedule. Unlike a right rotation, a right shift discards the bits shifted beyond the right boundary and fills the vacant positions on the left with zeros. The message-schedule functions are defined as

$$\sigma_0(x) = \text{ROTR}^7(x) \oplus \text{ROTR}^{18}(x) \oplus \text{SHR}^3(x)$$

$$\sigma_1(x) = \text{ROTR}^{17}(x) \oplus \text{ROTR}^{19}(x) \oplus \text{SHR}^{10}(x)$$

These functions expand the original sixteen 32-bit words of a message block into the 64 words $W_0, \dots, W_{63}$ required by the 64 rounds:

$$W_t = \begin{cases} M_t & 0 \leq t \leq 15 \\ \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16} \pmod{2^{32}} & 16 \leq t \leq 63 \end{cases}$$

After all 64 rounds have been completed, the resulting working variables are added modulo 2^{32} to the eight intermediate hash values that were used to initialize the block. The resulting state becomes the input state for the next message block.

After the final block has been processed, the eight 32-bit state values are concatenated to form the final 256-bit SHA-256 digest [8].

SHA-256 is designed to provide collision resistance, and this thesis relies on that property. A false cache hit caused by a cryptographic collision would be harmful in this system because it could cause the OCR result of one page to be associated with an unrelated page.

The following calculation concerns accidental collisions under an idealized random-output model; it does not measure resistance to a deliberate collision attack. Let N_{hash} denote the number of distinct canonical byte sequences hashed by the system. Assuming that the corresponding SHA-256 outputs behave as independent, uniformly distributed 256-bit values, the probability of at least one accidental collision is

$$P_{\text{coll}}(N_{\text{hash}}) = 1 - \prod_{k=0}^{N_{\text{hash}}-1} \left(1 - \frac{k}{2^{256}}\right).$$

For $N_{\text{hash}} \ll 2^{128}$, the birthday-bound approximation is

$$P_{\text{coll}}(N_{\text{hash}}) \approx \frac{N_{\text{hash}}(N_{\text{hash}} - 1)}{2^{257}}.$$

Using all 93,497 observable page requests as an upper bound on N_{hash} gives the following conservative estimate:

$$P_{\text{coll}}(93,497) \approx \frac{93,497 \times 93,496}{2^{257}} \approx 3.77 \times 10^{-68}.$$

The audit records 69,462 cache entries, indicating that many observable page requests resolved to previously stored cache identities. However, because the canonical byte sequences are not retained, the exact number of distinct canonical byte sequences cannot be independently recovered from the aggregate snapshot. Under this idealized model, the probability of an accidental SHA-256 collision is therefore negligible at the scale of the evaluated workload.

The current implementation does not retain the canonical PNG bytes or perform a byte-wise comparison after a cache-key match. When the SHA-256 digest, canonicalization version (`RENDER_VERSION`), and OCR model identifier are equal, the pages are treated as having the same cache identity. Consequently, a genuine SHA-256 collision would be handled as a cache hit and could not be distinguished from an ordinary repeated representation using the current cache.

Technical
University of
Denmark

Lautrupvang 15
2750 Ballerup, Denmark
Tlf. 4525 1700

<https://engtech.dtu.dk/>