**Eigentokens: Grammar‑Aware Inline Deduplication and Range‑Friendly Object Storage for AI/Analytics**

*If LLM models are build handcrafted in assembly now, I must confess to have invented a modular language to already compile them.*

Benjamin‑Elias Probst

Diplom Informatik (Examination Reg. 2010) — Profile Project (Applied Research)

Technische Universität Dresden — Chair of Scalable Software Architectures for Data Analytics (Prof. Dr. Michael Färber)

Supervision requested: Prof. Dr. Michael Färber

2025-10-08 — Winter Term 2025/26

E‑mail: benjamin‑elias.probst@mailbox.tu-dresden.de, benjamineliasprobst@gmail.com | Tel.: +49 162 327 8627

# Abstract

We propose **Eigentokens**, a storage‑internal grammar tokenization rule-scheme, enabling LLM grammar in storage, structuring and adaptation, by using asynchronous inline deduplication and lossless compression and preserving range‑friendly reads in an S3/KV object interface, as well as an additional LLM analysis interface. The grammar is constructed heuristically from byte‑stream similarity and mapped to a non‑strict B+ forest to align tokens with seekable blocks and offsets. The project evaluates dedup and context efficiency, ingest throughput, write amplification, and HTTP Range latency versus CDC baselines.

# Preface

Eigentokens and the resulting B+-forest-ELM (Eigentoken language model) completely go without floating point operations to be either correct or incorrect on their knowledge detection. The goal is to self-analyze any unknown grammar into organized shards and subprograms without supervision, to create language models, whilst storing them in an efficient compressed form at the same time. So, ELMs are metamodeling programs in M2, that can be part of metamodel M3 cluster machinery for self-aware systems. Instead of building a model they give a place to define how a model in M1 can be defined by M2 rules. Eigentokens in ELMs allow the construction of truly deterministic LLMs that only reside in polynomial time.

Core thought of the innovation from the perspective of the machine processing Eigentoken language modeling in M2 metalevel (ELM2):

1. Be the model
2. Understand yourself by “thinking” every thought possible from your knowledge
3. Organize your knowledge and train hard
4. Know it all or nothing
5. Describe yourself to others and your thought process (M3 autonomous self-metamodeling by using M2 model grammar)

Comment from the author: Knowing something or deciding if something is known to a probability is called dementia or hallucinations, failing to self-reflect knowledge is called low self-awareness. Splitting knowledge from the process of learning new things is called dogmatism. Failing to adapt to changes may be a form of autism. So, in terms of computers, it may be that LLM systems these days carry many sicknesses in terms of unfulfilled goals and non-available self-development without human intervention. To introduce human-like learning, we need to introduce human-like processing of data, accessing knowledge deterministically, but creating knowledge by interconnection of data until an efficient point we chose (virtually deterministic, but terminated in reality).

# Motivation

Modern AI/analytics pipelines exhibit high redundancy and frequent small range reads. Conventional object stores either compress whole blobs (hurting seekability) or deduplicate coarsely (losing edit‑locality). Learning and training AI models is mostly still a costly manual task. Learning, training, storing and gathering data are still split disciplines. This project proposes **Eigentokens**—storage‑internal grammar tokens, layed out as byte fragments, that reorganize similar byte snippets into a non‑strict B+ forest. Incoming objects are asynchronously inline deduplicated and losslessly compressed; similar snippets are arranged into grammar productions that remind us of eigenvector production and mathematical matrix reproducibility. Eigentokens reorganize similar byte fragments into grammar productions and lays them out as a non‑strict B+ forest, enabling token‑aligned range reads without full rehydration. Tokens are not linguistic words, but arbitrary byte‑stream segments constructed heuristically due to storage‑level pressures (dedup boundaries, range I/O). Externally, an S3/KV facade exposes objects and fingerprints. The aim is to reduce storage and I/O costs without sacrificing random‑access latency or operational simplicity. The target of organizing storage in this way is saving storage, while making the storage become the poly-time omni-model. The aim is to reduce storage and I/O costs for LLM processing, without sacrificing random‑access latency or operational simplicity.

We have observed within ourselves, that humans’ intelligence is based on learning sequential tasks and recombining them recursively by ordering multiple experiences as single blocks or groups of experiences, while base data is interpreted from different perspectives. Forming our knowledge is a modular process, therefore learning itself as a concept should be. Evolving AI technologies into a modular science is a breakpoint software engineering already has seen. In the result we may see an evolution that AI technologies may set themselves apart as knowledge technologies, which have the fundament of computer science as their base, just as electrical engineering set apart computer science from itself.

In general, we have observed how the era of computer science evolved in terms of software engineering, unraveling assembly code into modular component structures using different compilers to transfer the plan of a programming language into a machine-readable description. Once more, I’d like to contribute to this procedure in terms of LLM Model creation, composition and research. Therefore, ELM models should create a new M2 meta-metamodel layer, that can compile knowledge into M1 language models, while being maintainable in transient components. Like C++ and Java delivered advantages to languages like C, Fortran, Cobol and Algol 60, in terms of modularity, concepts, build, templates and later design patterns, we now skip the process of reinventing the wheel to repeat the process on LLM evolution stepwise. ELM or CELM should be compound data models that live in the M2 meta layer and define LLM model construction by separating Interpretations and knowledge, like program context was once split away from the data to be processed on the Harvard architecture. For this summary we first require the fundaments of “Computer architecture” by Andrew S. Tanenbaum and compiler construction in “Compilers Principles, Techniques, and Tools Alfred V. Aho” to describe the processing of rules into detected patterns and behaviors, to build interpretations/programs on Eigentoken data. Second, we require the fundaments of “Invasive Software Composition” by Uwe Aßmann to define, merge, plug and replace data component that are detected within loosely stored object storages.

Nowadays we are still in the age of handcrafted LLM experiments and experience and we shall go the same way as software engineering did to reach maturity of knowledge science, which uses AI machinery as a tool of discovery.

# Prior Work: UltiHash vs. Eigentokens

This section clarifies how the proposed work by Benjamin-Elias Probst differs from his earlier prototype at UltiHash.

* UltiHash (earlier approach):
* Static chunking with a few brute‑force entry points into deduplicable snippets.
* Pre‑allocated 1‑GB blocks sorting static snippets; two‑level tree layout.
* Indirect deduplication up to ~40% observed, but no grammar, no range‑optimized meta‑structure.
* Eigentokens (this work):
* Grammar‑aware dynamic chunking: CDC seeding followed by merging into grammar tokens (productions) that remain stable under local edits (insert/shift/rename).
* Non‑strict B+ forest as layout/metadata image of the grammar (production = internal node, leaf = token‑aligned (e.g., zstd‑compressed) block with offsets).
* Asynchronous inline pipeline: immediately writable stable references; grammar and compression finalized in the background → reduced write amplification at high ingest.
* S3/KV facade with token‑aligned range maps for low tail latencies.

# Scope & Non‑Overlap

This work does not propose a new text tokenizer for Large Language Models (LLMs) and does not perform NLP evaluation. Eigentokens are storage‑internal clear byte‑stream grammar tokens for deduplication, lossless compression and layout. Overall Eigentokens deliver a fundament of the construction of deterministic ELMs. The evaluation focuses on system metrics (space efficiency, ingest throughput, write amplification, and HTTP Range read latency), not NLP quality metrics. Instead of modeling a grammar directly, Eigentokens are designed to machine learn an M1 metamodel to construct a metamodel grammar to then define how a grammar – tokenized or not – should be learned by the object storage. This leads to the creation of grammar to guide storage rules instead of learning LLM grammar from any language. Therefore, the machine learning strategy is inverse. The machine learning component governs grammar construction for storage layout, not linguistic modeling; hence also the learning strategy is inverse to LLM tokenization, since the data will create its grammar using autonomous M1 and M2 Metamodel construction.

# Problem Statement & Research Questions

* **RQ1 (Chunking & Grammar):** Can grammar‑aware dynamic chunking with grammar-creation machine learning (Eigentokens) improve deduplication ratio and edit locality versus state‑of‑the‑art Content‑Defined Chunking (CDC) under realistic edits (insert/shift/rename)?
* **RQ2 (Index & Layout):** Does mapping grammar structure to a non‑strict B+ forest reduce write amplification and improve range‑read latency versus flat object layouts or LSM‑style indirections?
* **RQ3 (Inline Pipeline):** What is the latency/throughput trade‑off of asynchronous inline deduplication and compression during ingest compared to offline pipelines?
* **RQ4 (Range Semantics):** Can we maintain seekability (P50/P95/P99 HTTP Range read latency for varying spans) on compressed/deduplicated objects comparable to uncompressed baselines during write, read, and delete?

RQP (Roadmap): How do replication/erasure policies behave when grammar leaves act as the unit of placement? How can we introduce a generative ELM modeling machine by using Eigentokens? (Beyond the first paper’s scope.)

# Related Work (concise)

* **Content‑Defined Chunking (CDC):** Rabin/gear‑hash, FastCDC and successors as baselines for boundary stability and dedup efficiency.
* **Grammar‑based compression:** Sequitur/Re‑Pair lineage; adapted here for storage layout rather than linguistic modeling.
* **Key‑Value/Object stores:** B+‑trees versus LSM‑trees — trade‑offs in write amplification, compaction, and recovery.
* **Range‑friendly compression:** Offset‑addressable compressed blocks and block maps for efficient HTTP Range reads.

# Related Work and Gap

Content‑Defined Chunking (CDC)—including FastCDC and successors—provides robust boundaries and good deduplication, but models neither hierarchical structure nor productions (no grammar), and offers no explicitly range‑optimized structure across multiple objects.

Grammar‑based compression (e.g., Sequitur/Re‑Pair; Straight‑Line Programs (SLPs) with self‑indexing) supports substring extraction on compressed data but is not designed as an object‑store layout with inline deduplication or S3‑level range semantics.

Seekable block formats (e.g., blocked GZIP/BGZF) are range‑friendly yet lack cross‑object deduplication and do not exploit grammatical reuse. Deduplication and compression thrive on an increase of reference knowledge. Keeping the accessible knowledge base on a maximum will increase the potential of deduplication for the trade of some asynchronous performance and more required implementation optimization.

Machine‑learning‑assisted autonomous AI chunking and rule/cookbook innovation can improve boundaries and resemblance detection, but prior work does not elevate a learned grammar to the primary layout structure of an object store.

Gap: No integrated system combines agentic grammar‑aware chunking, a range‑optimized B+‑forest layout, and an asynchronous inline pipeline within a single S3/KV object store and grammar cookbooks, that learn their own grammar from unknown sources across all available different objects in a database storage.

Now we will go into the state of the art by giving an example of how the 3 largest LLM families are built, trained and operated

# Approach & System Design (Known vs. Novel)

To highlight the novelty of the Eigentokens approach, it is instructive to contrast it with the **state-of-the-art large language model (LLM) architectures** dominating AI today. Below we overview three prominent model families and their design philosophies – which rely on **probabilistic, sub-symbolic** representations – and then explain how Eigentokens fundamentally differs with a **deterministic, grammar-based** paradigm.

* **OpenAI GPT-4.1 and GPT-5 (GPT Series):** *Architecture:* The GPT family are giant Transformer-based networks trained on vast corpora of text (and code, plus images in GPT-4) using next-token prediction. GPT-4.1, an enhanced iteration of GPT-4, maintains a dense decoder-only transformer architecture with hundreds of billions of parameters (exact figures proprietary), refined via extensive fine-tuning and reinforcement learning from human feedback (RLHF). GPT-5 (2025) introduced a **unified dual-model system**: a fast, efficient sub-model handles simple queries, while a deeper “**GPT-5 Thinking**” model is invoked for complex problems, with a learned router deciding between them based on context. *Training Data & Operation:* These models ingest internet-scale data (web text, literature, code, etc.), thereby encoding a broad range of knowledge implicitly in their weight matrices. They operate by computing probability distributions over the next token in a sequence, effectively **modeling language statistically** rather than via explicit rules. *Limitations:* GPT models have a fixed (though growing) context window (e.g. tens of thousands of tokens), and no built-in long-term memory beyond what’s compressed in the weights or provided in prompts. Their reasoning is **sub-symbolic** – they cannot cleanly separate “facts” or formal rules, and often **hallucinate** or produce inconsistent outputs if prompted beyond their learned statistical patterns. Even GPT-5’s advanced architecture (with its internal “thinking” mode) remains fundamentally a probabilistic sequence model; it improves speed and reasoning depth but does not incorporate explicit semantic or grammatical modules.
* **Google Gemini 1.5 (Pro/Flash):** *Architecture:* Gemini is a family of **multimodal** LLMs developed by Google DeepMind, succeeding models like PaLM 2 and LaMDA. Version 1.5 (early 2024) came in two main variants: **Pro** (a high-capacity model) and **Flash** (a faster, lightweight model). Gemini’s architecture builds on Transformer foundations but with **Mixture-of-Experts (MoE)** and advanced parallelism to scale up capabilities. Notably, **Gemini-1.5-Pro** offered an unprecedented context window on the order of *1 million tokens*, enabled by specialized attention mechanisms and external memory management, allowing it to ingest extremely large documents or even video frames as text. *Training & Design:* Gemini was trained on a **diverse, multimodal dataset** – not just text and code, but images, audio, and video transcripts – aiming to imbue the model with agent-like problem solving and tool use. It can break down tasks into intermediate “thought” steps (exposed in a *Flash* mode that shows its reasoning chain) and interface with external tools (e.g. search, calculators) as part of its responses. Despite these innovations, Gemini’s knowledge and skills are still learned through pattern recognition across its training data. *Limitations:* Like other LLMs, Gemini **lacks explicit symbolic representations** – it cannot natively create or follow formal grammar rules, it only emulates them through statistical learning. The complexity of techniques like MoE and huge context windows improves performance but also makes the model a massive black box requiring immense computational resources. It remains prone to errors such as contradictory or inaccurate outputs (hallucinations) when confronted with scenarios outside its training distribution, since it doesn’t **encode ground truth rules** – only correlations. In short, Gemini extends the probabilistic LLM approach to new modalities and scales, but **does not depart from the probabilistic paradigm**.
* **Anthropic Claude 3.5 *“Sonnet”***: *Architecture:* Claude 3.5 (introduced mid-2024) is Anthropic’s latest large language model, focused on efficiency and alignment. It uses a Transformer-based architecture similar to GPT, trained on a massive text and code corpus, and notably expanded the context window to **~200,000 tokens** to handle very large inputs. Through careful engineering and likely model compression/distillation, Claude 3.5 achieves **roughly 2× the speed** of its predecessor (Claude 3 “Opus”) while improving performance on complex tasks. It also incorporates **vision capabilities**, able to interpret images and charts, making it multi-modal to an extent. *Training & Design:* Anthropic trained Claude with a special emphasis on **“Constitutional AI”** – instead of relying solely on human feedback to fine-tune behavior, they defined a set of guiding principles (a “constitution”) that the model uses to self-supervise and refine its outputs for harmlessness and coherence. Operationally, Claude 3.5 is offered at different tiers (e.g. instant vs. improved versions), but **all operate as probabilistic text generators** under the hood. *Limitations:* Claude 3.5, despite some unique alignment methodology, is **still a probabilistic LLM** without transparent internal logic. It doesn’t possess a built-in knowledge graph or rule system; all knowledge is stored as implicit connections in its neural weights. Thus, it can still **produce incorrect statements or reasoning** if prompted adversarially or if it encounters gaps in its training familiarity. Its large context window mitigates some memory limitations by allowing more reference text, but this is a workaround rather than a true long-term symbolic memory. The model’s improved safety is achieved by additional training constraints, not by introducing explicit rules or logic circuits. In summary, Claude 3.5 exemplifies a highly optimized **neurosymbolic** model (neural at core with some higher-level guidance), yet it remains **firmly on the probabilistic side** of the spectrum, without the deterministic, modular knowledge representations that a truly symbolic system would have.

# Eigentokens: A Novel Deterministic Grammar-Based Paradigm

In contrast to the above, **Eigentokens** takes a fundamentally different approach to representing and manipulating information. It is a **deterministic, grammar-inducing system** rather than a probabilistic neural network. Instead of adjusting millions of weights to **statistically approximate** a language or data distribution, Eigentokens explicitly **learns a grammar** from the data. This involves a meta-learning process: an **M1 metamodel** first learns how to construct a *metamodel grammar* for the incoming data streams (i.e. the system learns *how to learn* the grammar). The outcome of this process is a set of **explicit production rules** that can exactly regenerate segments of the data and are the interpretation part of the stored objects and shards, called Eigentokens. In other words, Eigentokens create the fundament to produce a *formal grammar* tailored to the dataset, capturing repetitive structures and patterns as reusable modularized objects, patterns and rules. This approach yields **lossless, interpretable representations**: each token and rule has a concrete definition (a sequence of bytes it expands to), unlike an LLM’s opaque embeddings. The system’s knowledge is thus **symbolically organized** as grammar rules, managed by the Eigentoken internals, which is the inverse of an LLM’s strategy — rather than burying the grammar of the data in millions of parameters, Eigentokens **derives the grammar directly** and stores it transparently.

Crucially, Eigentokens tokens and rules behave like **modular building blocks** of knowledge. The learned grammar can be seen as a **“cookbook” of rules** describing the dataset that is analog to the representation of grammar: each rule (or *knowledge module*) is a recipe that the storage-LLM/ELM engine can use to reconstruct a certain pattern or sub-object. These modules are stored in a **B+‑forest** (a collection of many flavors of B+‑tree indexes), which organizes the grammar productions and their occurrences in a way that supports efficient lookup and assembly. This design is analogous to **modular software composition** in classical software engineering (as explored by Prof. Uwe Aßmann et al.), extending it with the possibility of M3 self-adaptation on ELMs: just as software is built from modules or libraries that encapsulate certain functionality, Eigentokens build data representations from self-contained grammar components. Each component (grammar rule) is **autonomously self-descriptive** – it explicitly defines the content it represents and can be understood in isolation (e.g. a rule might say “Token\_42 = <common byte sequence>”). There is no mystery as to what a given Eigentoken means or contains. By combining these modules, the system can construct complex objects in a **compositional, deterministic** manner (much like linking together software modules), as opposed to an LLM’s diffuse generation process. This modularity not only improves interpretability but also means the system’s behavior is driven by **structured rules** rather than probabilistic inference.

While our project is focused on storage efficiency and data management, the **principles of Eigentokens hint at a broader AI capability**. By converting raw data into grammar-based knowledge modules, we lay a foundation for a future **generative system with symbolic traits**. In principle, an Eigentokens-powered ELM engine could be extended to function as an *omni-LLM* – a model capable of generating outputs (text, code, etc.) using its grammar modules instead of neural activations. Such a system would generate new sentences or data by **executing the production rules** (in novel combinations or sequences) rather than by sampling from a neural probability distribution. This would imbue the generation process with **logical consistency and traceability reducing cost**: the origin of each generated token could be traced back to a rule in the knowledge base, creating a solid and maintainable reasoning fundament, much like how a compiler expands macros or functions. Outcome data is therefore dependent on the bugs set on the input, but debugging data by deleting and modifying tokens is thinkable. One could imagine the Eigentokens knowledge base growing and updating autonomously as new data comes in – an AI that **self-describes and self-organizes** its knowledge in grammars, potentially mitigating issues like hallucination because it *knows exactly* which rules it’s applying. Base data can either be correct or incorrect, existent or not. Achieving an AI that combines LLM-like versatility with strict symbolic grounding is an ambitious vision (beyond the scope of this storage project), but the **Eigentokens approach marks a step in that direction**. By prioritizing explicit structure over statistical guesswork, our system design moves toward bridging probabilistic and symbolic methods in AI into becoming a mature engineering discipline.

**In summary,** Eigentokens diverge from conventional LLM architectures by using **deterministic grammar rules** as the core representation of knowledge. Below, we outline the concrete system design and components that implement this novel approach:

A1 – **Eigentokens & Dynamic Chunking:** Seed chunk boundaries via CDC; refine them into grammar tokens by merging similar snippets into productions stable under local edits. Maintain token IDs and one or more object fingerprints; allow recursive subdivision for hot ranges.

A2 – **B+‑Forest Metadata:** Map the grammar to a non‑strict B+ forest—internal nodes encode productions; leaves hold raw or pre‑processed (e.g., zstd‑compressed) snippets. Preserve order and offsets to support range reads without full rehydration.

A3 – **Asynchronous Inline Pipeline:** Ingest computes similarity and grammar updates asynchronously; stable references are written immediately; background tasks finalize compression and index compaction.

A4 – **API & Integration:** S3‑compatible object interface; KV semantics; per‑object fingerprint export. Range GET is served via token‑aligned block maps for efficient partial reads; optional BATCH GET for batched data loader access.

A5 – **Analysis/Control API:** To get metrics from the system to grammars, patterns and rules, be require a second interface to set database behavior.

A6 – **Eigentoken mock of local rules:** Develop some pattern rules for Eigentoken processing to demonstrate storage and grammar linking behavior.

A7 – **(Roadmap) Replication/Erasure:** Apply placement and erasure-coding policies on grammar leaves for resilience and space efficiency (future extension beyond the first project scope).

# Contributions

* C1 (Algorithmic): Grammar‑aware dynamic chunking improves deduplication and edit localization versus CDC under insert/shift/rename workloads.
* C2 (Systems/Layout): A non‑strict B+ forest over productions reduces write amplification and maintains stable range latencies (P50/P95/P99).
* C3 (Pipeline): Asynchronous inline deduplication and compression achieve near‑baseline ingest throughput while improving space efficiency.
* C4 (Benchmarking): An open harness with FastCDC, fixed‑size chunking, and optionally LSM as baselines; ablations removing grammar, forest, or async components.

# Comparative Landscape (including UltiHash)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Approach | Cross‑Object Dedup | Edit Stability | Range Reads | Write Ampl. | Ingest | Metadata | Layout |
| Fixed‑Size + zstd | Medium | Low | Medium (block map) | Low | High | Low | Flat |
| CDC / FastCDC | High | High | Medium (block map) | Medium | High | Low–Medium | Flat/LSM |
| Grammar + Self‑Index (SLP/Sequitur/Re‑Pair) | High (intra‑object) | Medium | High (substring) | High | Low–Medium | High | Index‑centric |
| Seekable Block (BGZF etc.) | n/a | n/a | High | Low | High | Low | Block‑map |
| UltiHash (earlier) | Medium (indirect) | Low–Medium | Medium | Medium | Medium | Low | 2‑level, static |
| Eigentokens (this work) | High | High | High (token‑aligned) | Lower | High (async) | Medium | B+‑forest |

# Evaluation Plan

Datasets:

* Code corpora (high near-deduplicate rate)
* Text corpora / logs (append-heavy, frequent edits)
* Columnar blobls (e.g. Parquet/CSV) typical for analytics
* Synthetic edit workloads (insert/shift/rename) to stress boundary stability

Baselines:

* Fixed‑size chunking with standard compression (e.g., zstd)
* CDC (Rabin, FastCDC, brute-force hashing) with and without compression
* Flat object layout without grammar mapping
* Optional: LSM‑style index for comparison (if time permits)

Metrics

* Space: deduplication ratio, compression ratio, index size
* I/O: ingest throughput, write and delete amplification; read latency (P50/P95/P99) for HTTP Range GET requests across cold/warm cache
  + Shard-switching related I/O delay measurement
* Compute: CPU‑seconds per GB processed, memory overhead
* Robustness: edit locality under shifts; crash/recovery behavior; index rebuild time

Datasets: code corpora (near‑duplicates), text/log corpora (append‑heavy), columnar blobs typical for analytics (e.g., Parquet/CSV), and synthetic edit workloads (insert/shift/rename).

Baselines: fixed‑size+zstd, CDC (Rabin/FastCDC) with/without compression, flat layout without grammar mapping, and optionally an LSM‑style index.

Metrics: space (deduplication and compression ratios; index footprint); I/O (ingest throughput; write/delete amplification; HTTP Range latencies P50/P95/P99 across cold/warm caches; shard‑switching delay); compute (CPU‑seconds/GB; memory); robustness (edit locality under shifts; crash/recovery; index rebuild).

# Timeline (15 weeks, indicative)

| **Phase** | **Milestones** |
| --- | --- |
| **W1–W2** | Finalize spec and architecture; micro-design of Eigentokens & B+‑forest; benchmark harness and metric interface. |
| **W3–W8** | Prototype ingest path (CDC → grammar induction → dedup index → compression); implement fingerprint export; basic S3 facade (PUT/GET/HEAD). |
| **W9–W10** | Implement range read path & token-aligned block maps; performance tuning; ensure crash‑safe metadata. |
| **W11–W12** | Evaluation runs (benchmark scenarios, ablation studies, plotting); draft preliminary results. |
| **W13–W14** | Writing (compose paper‑style report and finalize Exposé). |
| **W15–W20** | Buffer period for refinement, additional experiments, and submission planning (targeting a workshop or short-paper venue). |

# Risks & Mitigations

* **Grammar induction overheads** may impact ingest throughput — Mitigation: use an asynchronous pipeline and enforce a bounded tokenization depth to cap processing cost.
* **Range‑friendly mapping** could increase metadata size — Mitigation: employ token‑aligned block maps and compact leaf storage policies to limit metadata bloat.
* Implementation scope vs. semester time (risk of attempting too much in one term) — *Mitigation:* prioritize core components A1–A3; implement A4 as a minimal S3 subset (rather than full API) if needed; defer replication/EC (A7) to future work as planned.

# Assessment Alignment & Deliverables

* Colloquium (60 min): presentation, demo, and Q&A on design, evaluation, and implications.
* Deliverables: C++ prototype + CLI, reproducible benchmark scripts, datasets/pointers, report (PDF), slide deck (PDF), and a ~20‑page workshop‑style draft.
* Open benchmarking harness: ablations tied to research questions; transparent profiling and tail‑latency reporting.
* We evaluate deduplication efficiency, ingest throughput, write amplification, and HTTP Range latencies (P50/P95/P99) versus Content‑Defined Chunking (CDC) family baselines and flat/Log‑Structured Merge (LSM) layouts.

# Selected References

CDC & Deduplication: LBFS (SOSP’01), Venti (FAST’02), FastCDC (USENIX ATC’16) and later analyses.

Grammar‑based compression/self‑indexing: Sequitur, Re‑Pair, SLP surveys; grammar self‑indexes.

Indexes & KV/Object metadata: LSM trees, B/B+ trees, WiscKey, SILT, FASTER.

Range‑friendly access: HTTP Range (IETF), BGZF/Tabix, SAM/BAM toolchains.

Tokenizer literature for contrast only: BPE, WordPiece, SentencePiece/Unigram.

# Related work

**Content‑Defined Chunking (CDC) & Deduplication**

**[1] A. Muthitacharoen, B. Chen, and D. Mazières, “A Low‑Bandwidth Network File System (LBFS),” SOSP 2001. —** Foundational use of content‑defined chunking (CDC) for detecting similarity across file versions; establishes the CDC rationale used by many dedup systems. PDF: https://pdos.csail.mit.edu/papers/lbfs%3Asosp01/lbfs.pdf

**[2] S. Quinlan and S. Dorward, “Venti: A New Approach to Archival Storage,” FAST 2002. —** Classic content‑addressable, write‑once archival store; motivates fingerprint‑addressed blocks and global dedup indices. USENIX: https://www.usenix.org/conference/fast-02/venti-new-approach-archival-data-storage

**[3] W. Xia et al., “FastCDC: A Fast and Efficient Content‑Defined Chunking Approach for Data Deduplication,” USENIX ATC 2016. —** State‑of‑the‑art CDC variant (Gear hashing) balancing throughput and dedup ratio; baseline for modern CDC throughput/ratio trade‑offs. PDF: https://www.usenix.org/system/files/conference/atc16/atc16-paper-xia.pdf

**[4] Y. Hu et al., “The Design of Fast Content‑Defined Chunking for Data Deduplication,” IEEE TPDS, 2020. —** Journal extension analyzing FastCDC design decisions; useful for parameterization and performance modeling. PDF: https://ranger.uta.edu/~jiang/publication/Journals/2020/2020-IEEE-TPDS(Wen%20Xia).pdf

**[5] M. Gregoriadis et al., “A Thorough Investigation of Content‑Defined Chunking,” arXiv, 2024. —** Recent comparative analysis of CDC families; helpful as a survey for algorithm choices and distributions. arXiv: https://arxiv.org/pdf/2409.06066

**[6] M. O. Rabin, “Fingerprinting by Random Polynomials,” 1981 (Tech. Report). —** Origin of polynomial rolling fingerprints used in CDC and similarity detection. PDF: https://www.xmailserver.org/rabin.pdf

**Grammar‑Based Compression & Operating on Compressed Data**

**[7] C. G. Nevill‑Manning and I. H. Witten, “Identifying Hierarchical Structure in Sequences: A Linear‑Time Algorithm (SEQUITUR),” DCC 1997. —** Introduces grammar‑based compression via on‑line rule induction; conceptual basis for grammar tokens. arXiv: https://arxiv.org/abs/cs/9709102

**[8] C. G. Nevill‑Manning and I. H. Witten, “Compression and Explanation using Hierarchical Grammars,” The Computer Journal, 1997. —** Detailed exposition and evaluation of grammar induction for compression. PDF: https://ml.cms.waikato.ac.nz/publications/1997/NM-IHW-Compress97.pdf

**[9] N. J. Larsson and A. Moffat, “Off‑line Dictionary‑Based Compression (Re‑Pair),” Proc. IEEE, 2000. —** Efficient offline grammar construction (Re‑Pair); informs batch/async grammar building for storage backends. Abstract: https://people.eng.unimelb.edu.au/ammoffat/abstracts/lm00procieee.html

**[10] M. Lohrey, “Algorithmics on SLP‑Compressed Strings: A Survey,” 2012. —** Survey of algorithms over straight‑line programs (SLPs); relevant for operating on compressed data without full decompression. PDF: https://www.eti.uni-siegen.de/ti/veroeffentlichungen/12-survey.pdf

**[11] F. Claude and G. Navarro, “Self‑Indexed Grammar‑Based Compression,” Fundamenta Informaticae, 2011. —** Self‑indexing over grammar‑compressed data; informs seek/extract on grammatically stored objects. DOI: https://doi.org/10.3233/FI-2011-565

**Delta Encoding**

**[12] A. Tridgell and P. Mackerras, “The rsync Algorithm,” Tech. Report, 1996. —** Classic delta‑encoding with rolling checksums; informs external delta paths and similarity heuristics. PDF: https://www.andrew.cmu.edu/course/15-749/READINGS/required/cas/tridgell96.pdf

**Indexes & Key–Value/Object Metadata**

**[13] P. O’Neil et al., “The Log‑Structured Merge‑Tree (LSM‑Tree),” Acta Informatica, 1996. —** Baseline for log‑structured indices (e.g., KV/object metadata) and write‑optimized ingestion. PDF: https://dsf.berkeley.edu/cs286/papers/lsm-acta1996.pdf

**[14] R. Bayer and E. McCreight, “Organization and Maintenance of Large Ordered Indices (B‑Trees),” Acta Informatica, 1972. —** Canonical reference for B/B+‑trees; background for the non‑strict B+‑forest mapping in Eigentokens. PDF: https://infolab.usc.edu/csci585/Spring2010/den\_ar/indexing.pdf

**[15] L. Lu et al., “WiscKey: Separating Keys from Values in SSD‑Conscious Storage,” FAST 2016. —** Key/value separation to reduce write amplification; relevant to value‑log designs under compression/dedup. PDF: https://www.usenix.org/system/files/conference/fast16/fast16-papers-lu.pdf

**[16] H. Lim et al., “SILT: A Memory‑Efficient, High‑Performance Key‑Value Store,” SOSP 2011. —** Designs for flash‑backed KV with tiny indexes; relevant for dedup indices and fingerprint stores. PDF: https://www.cs.cmu.edu/~dga/papers/silt-sosp2011.pdf

**[17] B. Chandramouli et al., “FASTER: A Concurrent Key‑Value Store with In‑Place Updates,” SIGMOD 2018. —** Modern high‑throughput KV with hybrid log; informs concurrency and hot‑set handling on compressed backends. PDF: https://www.microsoft.com/en-us/research/uploads/prod/2018/03/faster-sigmod18.pdf

**Distributed File/Object Storage (for system context)**

**[18] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google File System,” SOSP 2003. —** Chunked, replicated file system and client‑driven range reads; background for “Google filesystem methodology”. Google Research: https://research.google/pubs/the-google-file-system/

**[19] S. A. Weil et al., “Ceph: A Scalable, High‑Performance Distributed File System,” OSDI 2006. —** Object/file storage with CRUSH placement; informs object‑centric sharding and metadata decoupling. ACM: https://dl.acm.org/doi/10.5555/1267308.1267330

**Range‑Friendly Access & Seekable Compression**

**[20] R. Fielding et al., “RFC 7233: HTTP/1.1 Range Requests,” IETF, 2014 (obsoleted in RFC 9110 semantics). —** Protocol basis for partial object retrieval; relevant to range‑friendly object access semantics. IETF: https://datatracker.ietf.org/doc/html/rfc7233

**[21] H. Li, “Tabix: fast retrieval of sequence features from generic TAB‑delimited files,” Bioinformatics, 2011. —** Uses BGZF (blocked GZIP) indices for random access into compressed files; a mature pattern for seekable compression. Oxford: https://academic.oup.com/bioinformatics/article/27/5/718/279592

**[22] H. Li et al., “The Sequence Alignment/Map format and SAMtools,” Bioinformatics, 2009; and P. Danecek et al., GigaScience, 2021. —** Practical example of block‑compressed random access (BGZF) supporting range queries. https://academic.oup.com/bioinformatics/article/25/16/2078/204688 ; https://academic.oup.com/gigascience/article/10/2/giab008/6137722

**LLM Tokenization (for contrast/terminology)**

**[23] R. Sennrich, B. Haddow, and A. Birch, “Neural Machine Translation of Rare Words with Subword Units,” ACL 2016 (BPE). —** Canonical BPE tokenizer for NMT; contrasts with storage‑internal byte/fragment tokens. ACL: https://aclanthology.org/P16-1162/

**[24] Y. Wu et al., “Google’s Neural Machine Translation System,” arXiv 2016 (WordPiece). —** WordPiece subword units as production tokenizer; background for ‘tokenization’ in LLM/NMT contexts. arXiv: https://arxiv.org/abs/1609.08144

**[25] T. Kudo and J. Richardson, “SentencePiece,” EMNLP 2018 (System Demos). —** Language‑independent subword training (BPE/Unigram) from raw text; reference point distinct from Eigentokens. ACL: https://aclanthology.org/D18-2012/

**[26] T. Kudo, “Subword Regularization,” ACL 2018 (Unigram). —** Unigram LM tokenization; additional contrast to storage‑internal tokenization on bytes/blocks. ACL: https://aclanthology.org/P18-1007/

**References (full bibliographic details)**

[1] A. Muthitacharoen, B. Chen, and D. Mazières, “A Low‑Bandwidth Network File System,” Proc. 18th ACM SOSP, 2001.

[2] S. Quinlan and S. Dorward, “Venti: A New Approach to Archival Storage,” Proc. USENIX FAST, 2002.

[3] W. Xia, H. Jiang, D. Feng, and Y. Hua, “FastCDC: A Fast and Efficient Content‑Defined Chunking Approach for Data Deduplication,” Proc. USENIX ATC, 2016.

[4] Y. Hu, W. Xia, et al., “The Design of Fast Content‑Defined Chunking for Data Deduplication,” IEEE TPDS, 31(10), 2020.

[5] M. Gregoriadis et al., “A Thorough Investigation of Content‑Defined Chunking,” arXiv:2409.06066, 2024.

[6] M. O. Rabin, “Fingerprinting by Random Polynomials,” Harvard Univ., Tech. Rep., 1981.

[7] C. G. Nevill‑Manning and I. H. Witten, “Identifying Hierarchical Structure in Sequences: A Linear‑Time Algorithm,” Proc. DCC, 1997.

[8] C. G. Nevill‑Manning and I. H. Witten, “Compression and Explanation using Hierarchical Grammars,” The Computer Journal, 40(2‑3), 1997.

[9] N. J. Larsson and A. Moffat, “Off‑line Dictionary‑Based Compression,” Proceedings of the IEEE, 88(11), 2000.

[10] M. Lohrey, “Algorithmics on SLP‑Compressed Strings: A Survey,” 2012.

[11] F. Claude and G. Navarro, “Self‑Indexed Grammar‑Based Compression,” Fundamenta Informaticae, 111(1), 2011.

[12] A. Tridgell and P. Mackerras, “The rsync Algorithm,” Tech. Rep. TR‑CS‑96‑05, 1996.

[13] P. O’Neil, E. Cheng, D. Gawlick, and E. O’Neil, “The Log‑Structured Merge‑Tree (LSM‑Tree),” Acta Informatica, 33(4), 1996.

[14] R. Bayer and E. McCreight, “Organization and Maintenance of Large Ordered Indices,” Acta Informatica, 1(3), 1972.

[15] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci‑Dusseau, and R. H. Arpaci‑Dusseau, “WiscKey: Separating Keys from Values in SSD‑Conscious Storage,” Proc. USENIX FAST, 2016.

[16] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, “SILT: A Memory‑Efficient, High‑Performance Key‑Value Store,” Proc. ACM SOSP, 2011.

[17] B. Chandramouli et al., “FASTER: A Concurrent Key‑Value Store with In‑Place Updates,” Proc. ACM SIGMOD, 2018.

[18] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google File System,” Proc. ACM SOSP, 2003.

[19] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, “Ceph: A Scalable, High‑Performance Distributed File System,” Proc. OSDI, 2006.

[20] R. Fielding et al., “RFC 7233: Hypertext Transfer Protocol (HTTP/1.1): Range Requests,” IETF, 2014.

[21] H. Li, “Tabix: fast retrieval of sequence features from generic TAB‑delimited files,” Bioinformatics, 27(5):718–719, 2011.

[22] H. Li et al., “The Sequence Alignment/Map format and SAMtools,” Bioinformatics, 25(16):2078–2079, 2009; P. Danecek et al., “Twelve years of SAMtools and BCFtools,” GigaScience, 10(2):giab008, 2021.

[23] R. Sennrich, B. Haddow, and A. Birch, “Neural Machine Translation of Rare Words with Subword Units,” ACL 2016.

[24] Y. Wu et al., “Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation,” arXiv:1609.08144, 2016.

[25] T. Kudo and J. Richardson, “SentencePiece,” EMNLP 2018 (System Demonstrations).

[26] T. Kudo, “Subword Regularization,” ACL 2018.

[27] OpenAI. “GPT‑4o System Card.” (PDF). https://cdn.openai.com/gpt-4o-system-card.pdf

[28] OpenAI. “GPT‑4.1 now in the API.” (Product/announcement page). https://openai.com/

[29] University of Florida AI. “GPT‑4.1 overview.” https://docs.ai.it.ufl.edu/

[30] Google DeepMind. “Gemini 1.5: Unlocking multimodal long‑context understanding.” arXiv:2403.05530 (Technical Report). https://arxiv.org/abs/2403.05530

[31] Anthropic. “The Claude 3 Model Family — Model Card.” (PDF). https://www.anthropic.com/ (direct PDF: https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model\_Card\_Claude\_3.pdf)

[32] Anthropic. “Introducing Claude 3.5 Sonnet.” (Announcement & model card addendum). https://www.anthropic.com/