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 LLM 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

Glossary

ELM – Eigentoken Language Model(er)

CELM – Component-based Eigentoken Language Model(er)

Interpretation – a parameterized program to give at least one output stream or result data from at least one input stream or input data.

Knowledge – Deeply for correspondence analysed information across at least one object

Data – Plain mathematical Information that needs to be interpreted to define a meaning

Storage – Any form of local or distributed institution to store structured data objects/tokens

B+-Tree – Indices and Eigentoken B+-Tree similar to indices B+-Trees in MySQL, MariaDB, PostgreSQL

B+-Forest – Several topic- or version-oriented B+-Trees, most of the time in a database/storage context to only contain filtered content or Eigentokens that offer linkage to a set of well-defined topics

Model-bucket – As we gather data to learn knowledge by analysis, keeping data sorted and in order is key, for example we sort frog pictures into one bucket and kangaroos into another, text in a third bucket. Those buckets may have full sorted subcategories, recursively becoming increasingly fine granular. B+-Trees keep major versions of the machine itself referring to changes in the interpretation program, model-buckets keep order and versioning on the data side

Byte-grammar – A storage-near grammar over byte sequences (not linguistic words), used to form productions (rules) for reuse and layout.

Production (rule) – A named expansion that reproduces a byte sequence or composition thereof; forms the building blocks of Eigentokens.

Cross-object grammar – Grammar productions whose reuse spans multiple objects/files across the corpus.

Token-aligned block map – A range map aligning HTTP range offsets to token (grammar) boundaries for seekability.

Seekable compression – Blocked/offset-addressable compression (e.g., BGZF, zstd-seekable) enabling random access within compressed data.

CDC / FastCDC – Content-defined chunking with rolling hashes (e.g., Rabin, Gear); FastCDC is a modern variant optimizing throughput and ratio.

SLP – Straight-Line Program; a compact grammar representation used by grammar compressors and self-indexes.

RLZ (see Glossary) – Relative Lempel-Ziv; compression relative to a reference with fast random access. [37, 38]

Tail latency – High-percentile response latency (P95/P99), critical for storage read performance.

Write amplification – Extra I/O writes beyond logical data due to layout, compaction, or metadata updates.

Fingerprint – Content-derived identifier (e.g., hash) to address objects or chunks for deduplication.

S3/KV facade – An object/key-value API surface compatible with S3 semantics.

Asynchronous inline pipeline – Ingest writes stable references first, deferring grammar/consolidation to background tasks.

HTTP Range semantics (RFC 9110) – The current reference for partial content requests across HTTP versions; obsoletes RFC 7233.

zstd-seekable – A zstd framing/pointer approach allowing random access; practice via skippable frames/seek tables.

Eigentokens define a storage-internal, grammar-centric programming substrate that serves two purposes at once: (i) a deterministic, component-based language to assemble and compile Large Language Models (ELM/CELM), and (ii) a byte-grammar–aware storage kernel enabling inline deduplication and lossless compression with range-friendly access in an S3/KV facade.

Tokens are not linguistic words but reusable byte-level productions mapped onto a non-strict B+ forest; each token can carry data and/or an interpretation-program. This dual role turns the object store into a grammar-backed build system: Eigentokens can compile M2 meta-rules into M1 model artifacts while preserving seekability and space efficiency.

We evaluate both facets: (A) systems metrics (deduplication/compression ratios, ingest throughput, write amplification, and HTTP Range tail latencies P50/P95/P99) against CDC/BGZF/seekable baselines; and (B) compilation efficacy of ELM/CELM (module reuse, determinism, reproducibility, and debug-ability), without relying on floating-point nondeterminism in core decisions.

Eigentokens is a storage-internal, grammar-centric substrate that does two jobs at once: it is a deterministic, component-based language for composing and compiling LLM artefacts (ELM/CELM), and it is a byte-grammar–aware storage kernel with inline deduplication, lossless compression, and token-aligned range access behind an S3/KV facade.

Tokens here are not words. They are reusable byte-level productions that can carry data and micro-programs and are mapped to a non-strict B+ forest. The store becomes a compiler's intermediate representation and build graph: productions, versions, and dependencies are first-class and remain seekable on disk (HTTP Range per RFC 9110). [33]

We evaluate both sides: (i) systems metrics—dedup/compression ratios, ingest throughput, write amplification, and P50/P95/P99 tail latency versus fixed-size+zstd, FastCDC±zstd-seekable, and BGZF [1, 3–5, 21, 22]—and (ii) compilation efficacy—module reuse, determinism, and reproducibility—without floating-point nondeterminism in core decisions. (CDC/FastCDC; Sequitur/Re-Pair/SLP for contrast: [1, 3–5, 7–11].)

Abstract

Modern AI/analytics pipelines keep duplicating themselves: similar code snapshots, evolving logs, columnar blobs, tiny deltas everywhere. Small range reads dominate, while storage systems still force a trade-off: coarse dedup hurts locality; monolithic compression kills seekability. That's expensive—and it stays opaque for deterministic builds.

I want the opposite: a store that behaves like a compiler. Eigentokens treat recurring byte-patterns as reusable productions, remain seekable on disk, and let me compose artefacts deterministically—without the floating-point roulette. Dedup and compression are by-products of the same grammar. Range access stays first-class (HTTP Range, S3/KV). [20]

Baselines exist and they are strong: CDC/FastCDC for edit-stable boundaries, and BGZF/zstd-seekable for random access. But none of them turns storage into a build graph with cross-object grammar as the primary image. That is the gap I am closing. [1, 3, 4, 21, 22]

Motivation

Scope & Non-Overlap

Scope & Non-Overlap. Eigentokens does not define a new NLP tokenizer and no generative language model per se. It defines a storage-near **compilation language** for LLM artefacts over byte-grammars; linguistic tokenization is out of scope. Determinism claims are confined to storage-level grammar induction, layout, and build orchestration; probabilistic inference remains separate. Evaluation focuses on system metrics and compile-time reproducibility, not on NLP-benchmarks.

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 grammaraware dynamic chunking with grammar-creation machine learning (Eigentokens) improve deduplication ratio and edit locality versus stateoftheart Content-Defined Chunking (CDC) under realistic edits (insert/shift/rename)?

RQ1 (Chunking & Grammar): Can grammaraware dynamic chunking with grammar-creation machine learning (Eigentokens) improve deduplication ratio and edit locality versus stateoftheart ContentDefined 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/gearhash, FastCDC (see Glossary) and successors as baselines for boundary stability and dedup efficiency.

ContentDefined Chunking (CDC): Rabin/gearhash, FastCDC (see Glossary) and successors as baselines for boundary stability and dedup efficiency.

- Grammarbased compression: Sequitur/RePair lineage; adapted here for storage layout rather than linguistic modeling.
- Key-Value/Object stores: B+trees versus LSMtrees — tradeoffs in write amplification, compaction, and recovery.
- Range-friendly compression: Offsetaddressable compressed blocks and block maps for efficient HTTP Range reads.

KeyValue/Object stores: B+trees versus LSMtrees — tradeoffs in write amplification, compaction, and recovery.

Rangefriendly compression: Offsetaddressable 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 explicit range-optimized structure across multiple objects. [1, 3–5, 6]

Grammar-based compression (e.g., Sequitur/Re-Pair; Straight-Line Programs (SLPs (see Glossary)) 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.

Grammar-based compression (e.g., Sequitur/Re-Pair; Straight-Line Programs (SLP (see Glossary)s) 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 (see Glossary)) 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 cookbook, that learn their own grammar from unknown sources across all available different objects in a database storage.

Let's analyze first, how the 3 largest LLM families are built, trained and operated.

Reframing of scope. Eigentokens elevate prior 'grammar-aware deduplication' into a **deterministic compilation language for LLMs**. The same grammar tokens that power inline deduplication and range-friendly layout also act as **ELM/CELM modules** whose productions, versions, and dependencies are stored and resolved in a non-strict B+ forest. Thus, storage is not only the repository of data; it becomes the **compiler's IR and build graph** for constructing, analyzing, and debugging LLM artifacts.

Approach & System Design (Known vs. Novel)

Reframing. Eigentokens lifts 'grammar-aware deduplication' into a **deterministic compilation language**: the same byte-grammar tokens that stabilize boundaries also encode modules, interfaces, and builds in a non-strict B+ forest—a composition-driven view of system construction (cf. software composition). [33, 39]

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. [27–29]. [34]. [35]

- 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. [30]. [36]

- 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. [31, 32]

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 embedding. 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. [1, 3, 4, 7, 9, 10]

A2 – B+Forest Metadata: Map the grammar to a non-strict B+ forest—internal nodes encode productions; leaves hold raw or preprocessed (e.g., zstd-compressed) snippets. Preserve order and offsets to support range reads without full rehydration. [10, 14, 21, 22]

A3 – Asynchronous Inline Pipeline: Ingest computes similarity and grammar updates asynchronously; stable references are written immediately; background tasks finalize compression and index compaction. [15, 17]

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. [20]

A4 – API & Integration: S3compatible 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. [18, 19]

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).

C1 (Cross-Object Grammar Induction): A deterministic, storage-near grammar induction learns reusable productions across objects and keeps boundaries edit-stable under insert/shift/rename workloads.

C2 (Token-Aligned B+-Forest Layout): Productions are mapped to a non-strict B+-forest whose leaves are token-aligned and support seekable substring extraction without full rehydration.

C3 (Async Inline Ingest with Bounded Cost): An asynchronous inline pipeline writes stable references first and bounds tokenization depth to control CPU and write amplification.

C4 (Seekable Compression Unification): Grammar tokens unify with seekable compressed leaves (e.g., zstd-seekable/BGZF) to preserve random access while enabling cross-object reuse.

C5 (HTTP Range Compliance): An S3/KV path implements HTTP Range semantics per RFC 9110 with token-aligned block maps and correct HEAD/GET behavior.

C6 (Tail-Latency & Amplification Evidence): Empirical evidence of lower P95/P99 tail latencies and reduced write amplification versus fixed-size+zstd, FastCDC±zstd-seekable, BGZF, flat mapping, and (optionally) minimal LSM/WiscKey.

C7 (Ablation & Formal Properties): An ablation suite isolates grammar, forest, and async effects; invariants at boundaries and amplification bounds under edits are stated.

C8 (Deterministic Reproducibility): A deterministic C++ prototype and open harness with reproducible runs (datasets, scripts) without FP-nondeterminism in core decisions.

Contributions

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, LLM meta compiler

Evaluation Plan

Datasets:

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

Baselines:

- Fixedsize 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	Prototypes 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.

Bibliography

Content-Defined Chunking (CDC) & Deduplication

[1] A. Muthitacharoen, B. Chen, and D. Mazières, "A LowBandwidth Network File System (LBFS)," SOSP 2001. — Foundational use of contentdefined 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 contentaddressable, writeonce archival store; motivates fingerprintaddressed 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. — Stateoftheart CDC variant (Gear hashing) balancing throughput and dedup ratio; baseline for modern CDC throughput/ratio tradeoffs. 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

GrammarBased Compression & Operating on Compressed Data

[7] C. G. NevillManning and I. H. Witten, "Identifying Hierarchical Structure in Sequences: A LinearTime Algorithm (SEQUITUR)," DCC 1997. — Introduces grammarbased compression via online rule induction; conceptual basis for grammar tokens. arXiv: https://arxiv.org/abs/cs/9709102

[8] C. G. NevillManning 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, “Offline DictionaryBased Compression (RePair),” Proc. IEEE, 2000. — Efficient offline grammar construction (RePair); informs batch/async grammar building for storage backends. Abstract: https://people.eng.unimelb.edu.au/ammoffat/abstracts/lm00procieee.html

[10] M. Lohrey, “Algorithmics on SLPCompressed Strings: A Survey,” 2012. — Survey of algorithms over straightline 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, “SelfIndexed GrammarBased Compression,” Fundamenta Informaticae, 2011. — Selfindexing over grammarcompressed 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 deltaencoding 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 LogStructured MergeTree (LSMTree),” Acta Informatica, 1996. — Baseline for logstructured 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 (BTrees),” 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 SSDConscious Storage,” FAST 2016. — Key/value separation to reduce write amplification; relevant to valuelog designs under compression/dedup. PDF: https://www.usenix.org/system/files/conference/fast16/fast16-papers-lu.pdf

[16] H. Lim et al., “SILT: A MemoryEfficient, HighPerformance Key-Value Store,” SOSP 2011. — Designs for flashbacked 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 InPlace Updates,” SIGMOD 2018. — Modern highthroughput KV with hybrid log; informs concurrency and hotset 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 clientdriven 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, HighPerformance Distributed File System," OSDI 2006. — Object/file storage with CRUSH placement; informs objectcentric sharding and metadata decoupling. ACM: https://dl.acm.org/doi/10.5555/1267308.1267330

RangeFriendly Access & Seekable Compression

[20] R. Fielding et al., "RFC 7233: HTTP/1.1 Range Requests," IETF, 2014 (obsoleted in RFC 9110 (see Glossary) 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 TABdelimited 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 blockcompressed 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). — Languageindependent subword training (BPE/Unigram) from raw text; reference point distinct from Eigentokens. ACL: https://aclanthology.org/D18-2012/

[27] Anthropic. "Claude 3.7 Sonnet and Claude Code — Announcement," Feb 24, 2025. https://www.anthropic.com/news/claude-3-7-sonnet

[28] Anthropic. "Claude 3.7 Sonnet — System Card (PDF)," 2025. https://www.anthropic.com/claude-3-7-sonnet-system-card

[29] Anthropic. "Introducing Claude 4 — Opus 4 and Sonnet 4," May 22, 2025. https://www.anthropic.com/news/claude-4

[30] Anthropic. "System Card: Claude Opus 4 & Claude Sonnet 4 (PDF)," May 2025. https://www-cdn.anthropic.com/4263b940cabb546aa0e3283f35b686f4f3b2ff47.pdf

[31] Anthropic. “Introducing Claude Sonnet 4.5,” Sep 29, 2025. https://www.anthropic.com/news/claude-sonnet-4-5

[32] Anthropic. “Claude Sonnet 4.5 — System Card,” Sep 2025. https://www.anthropic.com/claude-sonnet-4-5-system-card

[33] OpenAI. “Hello GPT-4o (Omni),” May 13, 2024. https://openai.com/index/hello-gpt-4o/

[34] OpenAI. “Introducing GPT-4.1 in the API,” Apr 14, 2025. https://openai.com/index/gpt-4-1/

[35] OpenAI. “Introducing GPT-5,” Aug 7, 2025. https://openai.com/index/introducing-gpt-5/

[36] Gemini Team (Google). “Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context,” arXiv:2403.05530, 2024. https://arxiv.org/abs/2403.05530

[37] Kuruppu, S., Puglisi, S.J., Zobel, J. “Relative Lempel-Ziv Compression of Genomes for Large-Scale Storage and Retrieval,” SPIRE 2010. https://doi.org/10.1007/978-3-642-16321-0_20

[38] Kuruppu, S., Puglisi, S.J., Zobel, J. “Optimized Relative Lempel-Ziv Compression of Genomes,” CRPIT 2011. https://crpit.scem.westernsydney.edu.au/abstracts/CRPITV113Kuruppu.html

[39] U. Aßmann. “Invasive Software Composition.” Springer, 2003.

[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/