TECHNISCHE UNIVERSITÄT DRESDEN

CHAIR OF DATABASE TECHNOLOGIES
PROF. DR. MICHAEL FÄRBER

INF-PM-ANW Profil Anwendungsforschung in der Informatik

Eigentokens: Grammar-Aware Inline Deduplication and Deterministic Compilation for Large Language Models

Benjamin-Elias Probst
(Mat.-No.: 4510512)

Professor: Prof. Dr. Michael Färber
Tutor: Prof. Dr. Michael Färber

Dresden, February 21, 2026

Contents

Abstract

This report presents **Eigentokens**, a novel storage architecture that fundamentally reconceptualizes how Large Language Models (LLMs) are constructed and stored. Unlike contemporary probabilistic approaches that encode knowledge in opaque neural weights, Eigentokens introduce a deterministic, grammar-based compilation paradigm where storage operations and model construction are unified through explicit grammatical productions. The system operates on two integrated levels: (1) as a storage kernel providing grammar-aware inline deduplication with byte-level grammatical productions mapped into a non-strict B^+-forest index (SHA512 content-addressable [Dan15]), achieving superior compression ratios while maintaining seekable access via HTTP Range requests (RFC 9110[FNR22]); and (2) as a deterministic compiler for language models, where grammatical rules extracted from data directly determine neural network weights without any probabilistic training. Each *Eigentoken* is treated as a binary description language with an associated interpretation program, data payload, and references to other tokens, enabling hierarchical composition of knowledge. Through recursive grammar induction (with worst-case $O(n^3)$ complexity) and cross-object pattern learning, the system builds an extensible "grammar cookbook" of reusable rules that remain stable under edits.

Empirical goals for this C++ prototype include achieving **25–40% higher deduplication effectiveness** than state-of-the-art content-defined chunking (CDC) methods [MCM01]—in particular Fast content-defined chunking (FastCDC) [XZJ$^+$16]—on realistic datasets; maintaining **95th percentile (P95) read latencies within +15%** relative to an uncompressed storage baseline; and realizing **30% lower write amplification** compared to Log-Structured Merge-tree (LSM-tree) based storage systems [OCGO96]. The ultimate vision is an *omni large language model (omni-LLM)*: a universal model constructed through modular grammar execution rather than neural sampling, enabling complete determinism and traceability. Because output quality is bounded by input data quality, errors would propagate deterministically, but remain fully traceable and correctable. This report details the objectives, technical foundations, architecture, implementation, and planned evaluation of the Eigentokens system, and discusses expected results, limitations, and future outlook. In doing so, it focuses on two core questions: whether a language model can be compiled into a deterministic binary artifact and executed (RQ1), and what grammar is and how such grammar can be learned from data (RQ2).

Project Context and Assignment

Institution: Faculty of Computer Science, Institute of Systems Architecture, Chair of Scalable Software Architectures for Data Analytics, Technische Universität Dresden.
Degree program: Diploma in Computer Science (Prüfungsordnung (PO) 2010).
Module: INF-PM-FPA Profilprojekt Anwendungsforschung in der Informatik.
Work period: 28.10.2025 – 10.02.2026 (15 weeks).
Supervision and reviewers:

- Supervising professor and supervisor: Prof. Dr. Michael Färber
- First reviewer: Prof. Dr. Michael Färber
- Second reviewer: n. n.

Topic (condensed): Eigentokens: grammar-aware inline deduplication and a deterministic compilation language for Eigentoken Language Model (ELM) / CELM-lang (CELM) artifacts, using a Secure Hash Algorithm 512-bit (SHA-512) [Dan15] indexed non-strict B^+-forest layout (B-tree family [BM72]) and an S3/key-value (KV) facade with Hypertext Transfer Protocol (HTTP) Range semantics (Request for Comments (RFC) 9110) [FNR22].
Non-goals: Natural Language Processing (NLP) tokenization and probabilistic inference/sampling are out of scope for the initial prototype.

Work Packages (Merged)

- **Requirements and scope:** use-cases, Service Level Objectives (SLOs), terminology; explicit non-goals; evaluation plan.
- **Architecture and specification:** Eigentoken tuple $\langle id, P, D, R \rangle$, grammar induction algorithm, non-strict B^+-forest with SHA-512, token/block mapping, RFC-9110 compliant Range mapping.
- **Core implementation (C++):** Grammar Induction Engine (A1), B^+-Forest Index (A2), Asynchronous Processing Pipeline (A3), S3/KV Interface (A4), Analysis and Control Application Programming Interface (API) (A5), Mock Rules and Testing Suite (A6).
- **CELM-lang integration:** M2-level meta-language for interpretation programs, pattern generators, match conditions, dynamic runtime extension; foundations for M3-level self-improvement routines.
- **Range path and optimization:** token-aligned block maps, heuristics for hot ranges, asynchronous consolidation.
- **Benchmarks and evaluation (incl. ablations):** datasets (code, text/logs, binary, and columnar formats), baselines, metrics (space, ingest, write/deletion amplification, percentiles P50/P95/P99), robustness under shift/insert/rename transformations.
- **Artifacts and reproducibility:** Command Line Interface (CLI), scripts, reproducible plots, report (~20 pages), slide deck; scientific documentation and a colloquium presentation.
- **Roadmap (optional):** replication/erasure coding on grammar leaves; sketch the ELM/CELM compilation path for deterministic model generation.

Deliverables and Colloquium

Submission includes: C++ prototype with CLI, benchmark scripts (and datasets/links), reproducible evaluations (plots), report (Portable Document Format (PDF)), and slide deck (PDF). The project concludes with a colloquium featuring a 60-minute talk and serves as preparation for the diploma thesis.

1 Introduction and Motivation

Current large language models (LLMs) such as GPT-5 and Google Gemini are built on probabilistic neural architectures with billions of learned parameters. While powerful, these monolithic models have fundamental limitations: they act as opaque black boxes with no explicit declarative knowledge representation, are prone to hallucination, difficult to update with new information, and nearly impossible to debug or formally verify. When a model produces an incorrect statement, there is no straightforward way to trace that output back to a specific training datum or rule; correcting such errors often requires costly retraining on updated data. This lack of transparency and determinism in conventional LLMs poses a critical barrier for high-stakes applications that require explainability and reliability.

Deterministic Compilation of Large Language Models (LLMs): Eigentokens propose a paradigm shift by treating language model construction as a *compilation* problem rather than a probabilistic training problem. Instead of implicitly encoding knowledge in floating-point weights through gradient descent, the Eigentokens approach explicitly encodes knowledge as *grammatical production rules* extracted from a corpus. In essence, the system learns a formal grammar that describes the content of the data, and uses this grammar as a blueprint to compile deterministic neural network components. The storage layer and the model-building process are unified: storing data triggers grammar induction, and these learned grammar rules directly inform the structure and weights of a neural model (an Eigentoken Language Model (ELM)). This approach aims to provide:

- **Complete traceability:** Every component of a generated model (e.g., a weight or connection) can be traced back to specific source data patterns or rules that produced it.
- **Incremental and modular updates:** New knowledge can be integrated by adding or updating grammar rules (i.e., new tokens or productions) without retraining from scratch, enabling fast updates and fine-grained model editing.
- **Interpretability and debuggability:** The model's behavior is governed by human-readable rules (a "grammar cookbook") rather than inscrutable weights, making it possible to inspect, reason about, and verify the model's knowledge base.
- **Storage efficiency:** By leveraging grammar-based deduplication at the storage level, the system achieves significant data compression and deduplication improvements (targeting 25–40% better deduplication versus content-defined chunking (CDC) methods) while preserving fast random access to data.

Storage as AI Infrastructure: A key insight motivating this work is that effective storage deduplication is inherently linked to recognizing structure in the data—the very same structures that a language model could use to compose knowledge. Modern data lakes and versioned datasets exhibit enormous redundancy (e.g., repetitive code snippets, recurring log patterns, overlapping image regions). Traditional storage systems either sacrifice deduplication granularity for performance or vice versa. Eigentokens unify storage and model compilation: the storage engine performs *grammar induction* to deduplicate data, and those grammars become the building blocks of an LLM. In doing so, the storage system essentially doubles as the learning system, turning data redundancy into an asset for model construction. This is analogous to moving from low-level assembly (raw bytes) to higher-level languages (structured grammar) in software development—here, applying that evolution to AI model building.

2 Related Work and Background

2.1 Content-Defined Chunking and Deduplication

Our approach builds on a rich history of **content-defined chunking (CDC)** and deduplication techniques in storage systems. Classic systems like LBFS (Low-Bandwidth Filesystem) introduced CDC using rolling hash breakpoints to identify duplicate regions across file versions [MCM01]. Modern variants such as FastCDC improve throughput (processing data at gigabytes per second) and produce more stable chunk boundaries by using smarter hash triggering mechanisms [XZJ+16]. These CDC methods can eliminate 20–30% of redundant data on typical workloads, but they treat the data as an undifferentiated byte stream and lack any semantic or cross-object awareness. They also face a trade-off: smaller chunks improve deduplication but can bloat metadata and harm throughput, whereas larger chunks miss finer redundancies.

Eigentokens extend CDC by introducing **dynamic, grammar-aware chunk boundaries**. Instead of fixed-size or purely hash-based chunk splits, the grammar induction engine identifies variable-length patterns (from a few bytes to entire subdocuments) that can be abstracted as tokens. This approach is related to **grammar-based compression** techniques like *Sequitur*[NMW97] and *Re-Pair*, which form a context-free grammar by repeatedly replacing repeated substrings with nonterminal symbols. Those algorithms achieve excellent compression on single files by capturing hierarchical structure. However, traditional grammar compressors operate offline on individual sequences and are not designed for large-scale, cross-object scenarios or real-time storage systems. In contrast, Eigentokens performs online grammar induction across an entire corpus (cross-file), and integrates this with a persistent storage index and query interface.

Another line of relevant work is **self-indexed storage and compressed data structures** (e.g., using Straight-Line Programs and self-indexes) which allow queries on compressed data. These demonstrate that it is possible to operate directly on compressed representations. Eigentokens adopt a similar philosophy by making the grammar itself the index: data is stored in compressed form (via grammar rules), yet remains queryable via content hashes and references.

A closely related prior system is **UltiHash**, an earlier grammar-aware deduplication prototype by the same research group. UltiHash could detect repeated structures across a codebase (it successfully deduplicated 380k lines of code in a self-hosting experiment) and even scaled to industrial datasets (1 PB of video at Bosch). However, UltiHash was limited to storage deduplication; it did not incorporate deterministic model compilation. Eigentokens build upon UltiHash's storage optimizations and goes further by making the grammar the centerpiece of an LLM construction pipeline. In summary, no existing system combines:

- *Cross-object grammar induction* at scale (with $O(n^3)$ learning complexity heuristically tamed for practicality),
- *A B^+-forest index* optimized for grammatical and content-addressable access patterns,
- *Deterministic neural model compilation* directly from stored patterns,
- and an *agentic AI capability* where the storage system can self-improve by analyzing its own knowledge base.

This unique combination defines the research gap that Eigentokens aim to fill.

3 Technical Foundation

This section introduces the core concepts and formalisms underlying the Eigentokens architecture: the structure of an Eigentoken, the grammar induction process, the SHA512-indexed B^+-forest storage structure, and the *CELM-lang* meta-language used for interpretation programs.

3.1 The Eigentoken Model and Grammar Induction

At the heart of the system is the notion of an **Eigentoken**. In essence, an Eigentoken is a self-contained unit representing a piece of information (data) along with the method to interpret or reconstruct that information. Formally, each Eigentoken can be viewed as a tuple:

$$T = \langle id, P, D, R \rangle,$$

where id is a content-derived identifier (a Secure Hash Algorithm 512-bit (SHA-512) hash of the token's content by default), P is an *interpretation program* (the instructions or recipe for reconstructing the token's data), D is an optional raw data payload (literal content, present only for tokens that cannot or should not be decomposed further), and R is a set of references to other Eigentokens that P relies on (the token's dependencies). This design follows a Harvard-architecture-like separation: the program (P) is stored alongside the data (D) and references (R) it operates on. If D is `null`, the token is purely synthetic and must be built from the referenced tokens via program P.
From the perspective of formal language theory, the program component P can be viewed as a production rule: a grammar is a finite set of such rules that generates a language [Cho56]. Learning grammars from examples is known as grammar induction and is fundamentally challenging [Gol67]; Eigentokens therefore uses deterministic, engineering-oriented heuristics (Sequitur/Re-Pair-style inference [NMW97, LM00]) designed to remain stable under edits.
For example, if we have repetitive text data where the word "Apfelbaum" (German for "apple tree") appears multiple times, the system might create an Eigentoken representing "Apfelbaum" and decompose it into two smaller tokens "Apfel" and "baum". The composite token τ_3 for "Apfelbaum" could be stored as:

```
id: SHA512("Apfelbaum")
P: CONCAT(tau_1, tau_2)  # instructions: concatenate tau_1 and tau_2
D: null                  # no direct data, constructed from parts
R: { SHA512("Apfel"), SHA512("baum") }
```

Here τ_1 and τ_2 might be tokens for "Apfel" and "baum" respectively. Such a structure allows the data "Apfelbaum" to be reconstructed from smaller parts, which might be shared with other words like "Apfelsaft" or "Baumhaus", thereby reducing storage redundancy.
The process that produces these tokens is the **grammar induction algorithm**. It operates in multiple phases:

1. *Initial chunking:* As data is ingested, it is first divided into coarse chunks (using an approach akin to CDC to ensure content-defined boundaries). At this stage, chunks are stored as provisional tokens (with their raw data as D and trivial identity programs).

2. *Pattern discovery (Grammar induction):* In the background, the system scans the corpus for repeated byte substrings and sequences. It employs a hierarchical pattern discovery with worst-case $O(n^3)$ complexity (for input size n), though in practice this is mitigated by heuristics and the fact

that most data has structure that can be found in sub-quadratic time for each pattern. The algorithm is conceptually similar to Sequitur/Re-Pair, identifying the most frequent pair of adjacent symbols (bytes or existing tokens) and replacing them with a new rule, iteratively. However, it extends across all objects in storage (global grammar) and uses context-sensitive analysis to ensure the rules are meaningful across different files.

3. *Rule creation (Eigentoken formation):* Each discovered pattern results in the creation of a new Eigentoken (with a new id computed as the hash of the pattern bytes). Its interpretation program P is typically a composition of existing tokens (e.g., $\texttt{CONCAT}(\tau_i, \tau_j)$). The references R of this new token include τ_i and τ_j, and D is null because the token is defined purely in terms of smaller parts. These new tokens (grammar rules) are added to a global *grammar cookbook*. The induction algorithm continues iteratively, possibly discovering higher-level patterns that consist of these tokens, building a hierarchical grammar.

4. *Stability and generalization:* Importantly, the induction aims to find patterns that are *stable under edits* and common across multiple objects, not just within one file. For example, if a certain code snippet appears with slight variations across programs, the system might create tokens for the snippet and its variants, and then a higher-level token representing the common structure with parameters for differences. This cross-object generalization is a key differentiator from file-local compression: it treats the entire dataset as one large sequence for grammar discovery.

The outcome of grammar induction is a set of production rules (tokens) that form a generative description of the dataset. This set of rules, dubbed the **grammar cookbook**, serves a dual purpose: it is the basis for reconstructing any stored object (by following the tokens and their programs), and it is the knowledge base for compiling deterministic LLMs. Each Eigentoken captures a piece of "knowledge" (a pattern) that can be reused in many contexts. In total, the grammar cookbook essentially represents a compressed form of the data, akin to a highly factorized dataset.

3.2 SHA512-Indexed B$^+$-Forest for Token Storage

To store and organize the large number of Eigentokens produced, we introduce a **non-strict B$^+$-forest index** structure (from the B-tree family [BM72]) keyed by SHA-512 digests [Dan15]. A B$^+$-forest is essentially a collection of B$^+$-tree indexes that together index all tokens; each tree corresponds to a particular subset or view (for example, tokens related to a specific topic or data type). The use of multiple trees (a forest) allows logically partitioning the grammar by semantic category or by version, which can improve locality and query performance.
Key characteristics of the B$^+$-forest in Eigentokens include:

- **Content-addressable indexing:** The primary key for the index is the *id* of each token, which is a 512-bit SHA-512 fingerprint of the token's content or definition. This ensures global uniqueness and enables content-addressable storage: any time a token with the same content is inserted, it can be recognized and deduplicated by key match.

- **Non-strict, recursive structure:** Unlike a standard B$^+$-tree that stores totally ordered keys and associated values, our tokens may have references to each other forming a DAG. The index is *non-strict* in the sense that it allows references to other keys without requiring a strict hierarchy. We maintain referential integrity through careful update rules. The internal nodes of the B$^+$-trees can store summary information (like frequency counts of token usage, to inform the grammar learning heuristics), and the leaf nodes store either the raw data for base tokens or a pointer to the interpretation program plus references for composite tokens.

- **Topic-based filtering:** Each tree in the forest can correspond to a semantic category or model-bucket. For instance, one tree might index all tokens related to "code" data, another for "text",

another for "images", etc. This enables *topic-filtered views*: the system can quickly retrieve all tokens relevant for a given topic or domain by consulting the corresponding tree. This is particularly useful for compiling a model focused on a certain topic (see Section 5 on model compilation).

- **Range semantics and seekability:** To preserve the ability to do fast random access, we implement range semantics in the storage of tokens. This is analogous to maintaining an index for BGZF (blocked gzip) or other seekable compression formats, but here integrated with the grammar. The B^+-forest helps with this by storing the tokens in an order aligned with content (e.g., by token identifiers that preserve locality of references), allowing efficient mapping from logical byte ranges to sequences of tokens.

- **Scalability and distribution:** The forest can be sharded or distributed across nodes if needed (though the initial prototype is single-node). Write-ahead logging and snapshotting are used for crash consistency.

3.3 The CELM-Lang Meta-Language (Interpretation Programs)

We introduce **CELM-lang**, a custom meta-language developed for this purpose. It operates at the meta-model level (often referred to as M2 in model-driven architecture terms), meaning it is a language about how to process and compose tokens, rather than about the data domain itself.
CELM-lang provides a set of primitives tailored to describing token manipulation and composition. Key aspects include:

- It defines a set of **instructions and pattern combinators** that can be used in Eigentoken programs. For example, beyond simple concatenation, CELM-lang might include operations like `SPLIT(token, delimiter)`, `TOUPPER(token)` for case transformations, or `MATCH(regex)` for pattern matching. These operations enable tokens to be transformed or combined in flexible ways.

- CELM-lang rules can specify **match conditions** that guide the grammar induction. For instance, a rule could indicate that a certain pattern should only be applied in a specific context (e.g., a token that represents an XML tag might only match if a corresponding closing tag token is present). This allows domain knowledge or constraints to inform the induction process.

- CELM-lang can express **recursive and parameterized rules**, enabling the representation of entire classes of patterns with a single rule. This is useful for capturing structured data formats (like nested JSON patterns or XML trees) in a compact form.

- The language is designed to be easily extensible; it supports an **open-world assumption** where new instruction types can be added as needed to capture emerging patterns. This ensures the system is not limited to a fixed set of compressible structures.

By using CELM-lang to define token interpretations, we achieve a flexible **metaprogramming** capability: new types of patterns and transformations can be added without changing the C++ core of the system; instead, they are introduced as new CELM-lang constructs that the system's runtime can execute. This is analogous to how SQL or regex can extend what you can do with a database without recoding the database engine. In our case, CELM-lang enables the storage system to be taught new "analysis recipes" at runtime.
Furthermore, CELM-lang descriptions support the notion of an **omni-LLM** vision: because the language can describe interpretation programs that act on tokens, an advanced use is to have tokens that describe how to answer queries or how to modify the grammar itself. This is where *M3* (the self-adaptation metamodel) comes in: at the highest level, the system can use CELM-lang to script its own evolution. For example, an agentic query might trigger a CELM-lang routine that scans for inefficiencies in the

grammar (like two similar tokens that could be merged) and then introduces a new rule to improve it. This aspect is still exploratory, but it lays groundwork for a self-reflective AI system grounded in storage.

4 Design and Architecture

The Eigentokens system is comprised of several core components that together implement the grammar-based storage and compilation functionality. The architecture has been carefully designed to decouple the latency-sensitive storage operations from the heavier background analysis, and to provide interfaces for both data access and introspection/control. Figure (not shown) illustrates the high-level component architecture, which we describe below.

4.1 Core Components Overview

A1. Grammar Induction Engine: The component responsible for discovering patterns and creating new Eigentokens. It implements the multi-phase $O(n^3)$ grammar learning algorithm with numerous optimizations. In practice, it uses a three-phase approach: initial content-defined chunking, followed by a context-sensitive pattern mining across the corpus, and finally grammar rule consolidation. To handle large data, it employs *dynamic CDC* with grammar-aware boundaries (adjusting chunk splits on the fly when grammar patterns are recognized, rather than using fixed-size windows) and caches frequent sequences to avoid recomputation. This engine works mostly asynchronously in the background, but it hooks into the ingestion path to capture new data.

A2. B$^+$-Forest Index: The storage backend is a collection of B$^+$-trees, collectively a forest, as described in Section 3.2. This index stores all Eigentokens keyed by their SHA-512 hash. It supports operations to insert new tokens, look up tokens by id, and iterate or range-scan tokens (for range queries). The index is *non-strict* in that it is aware of the reference relationships between tokens, but it tolerates ephemeral inconsistencies (e.g., a token might reference another token that is not yet fully indexed, during intermediate states of analysis) knowing that eventual consistency will be reached once the induction completes. Each tree in the forest can be thought of as a shard or a filtered view (e.g., one tree per data domain or per top-level category). The B$^+$-forest provides $O(\log N)$ access time for tokens and scales to very large N due to the balanced tree structure.

A3. Asynchronous Processing Pipeline: Where beneficial, literal payloads can be stored in a seekable compressed form (e.g., zstd-seekable) to improve range-read performance without sacrificing compression ratio. To reconcile the need for immediate data writes with the expensive grammar analysis, Eigentokens employs an asynchronous multi-stage pipeline. When new data is ingested, the pipeline performs:

1. **Stage 1: Immediate Ingestion.** The data is chunked (content-defined) and each chunk is immediately assigned a provisional Eigentoken (with a hash id and raw data payload). These are inserted into the B$^+$-forest right away. This stage is optimized for low latency (target $<$10 ms per chunk for commit) to ensure that writes are acknowledged quickly. The output is a set of stable references (the new tokens or references to existing tokens if duplicates were found).
2. **Stage 2: Background Grammar Induction.** The Grammar Induction Engine (A1) picks up the new data from a work queue and begins analyzing it in the context of the global corpus. It performs a similarity search step (e.g., fingerprint- and reference-guided candidate retrieval) before deeper grammar induction to prioritize high-yield pattern candidates. It may discover that some chunks are redundant or part of larger patterns that have appeared before; it will create new higher-level tokens as needed. This stage runs concurrently and can take substantially longer (it might batch many updates together). The system ensures that Stage

1 output is always a correct (if not fully optimal) storage of the data, and Stage 2 will only refine the internal representation.

3. **Stage 3: Index Optimization and Compaction.** Periodically, the system rebalances and compresses the B^+-forest (similar to a compaction in LSM trees). It cleans up any redundant tokens that Stage 2 rendered obsolete (e.g., if a chunk got replaced by a grammar rule, the raw chunk might be dropped or archived) and updates reference links. This stage also integrates any deferred compression (if large literal payloads can now be compressed after identifying boundaries) and writes snapshots or checkpoints for recovery.

The pipeline decouples write latency from heavy processing: data is safely stored in Stage 1, and the later stages can lag behind without affecting data durability or immediate consistency. In case of a crash, the system can recover from the latest checkpoint plus the log of Stage 1 operations.

A4. Storage Interface (S3/KV Facade): Eigentokens exposes an interface compatible with object storage (like Amazon S3) and key-value store semantics, so that it can be used as a drop-in replacement for existing storage systems. The interface supports `PUT/GET/HEAD` semantics and can optionally accept client-provided fingerprints to short-circuit uploads when content is already present. Crash safety is ensured via write-ahead logging and periodic snapshots, such that Stage 1 commits remain durable independent of later consolidation. Applications can PUT and GET objects; under the hood these calls interact with the B^+-forest. The interface supports standard HTTP Range requests for partial reads [FNR22], thanks to the token-aligned block mapping (the system calculates which tokens correspond to the requested byte range and only reconstructs that portion). The KV interface allows direct addressing of tokens by hash as well, which is useful for deduplication (if a client provides a hash of content it is about to upload, the system can short-circuit and just return a reference if it already has that token). This layer also implements access control, basic metadata (size, timestamps), and could integrate with cloud storage APIs.

A5. Analysis and Control API: In addition to the data access interface, a secondary API is provided for introspection and control of the system. This includes retrieving statistics about deduplication (e.g., how many tokens exist, distribution of token sizes), querying the grammar cookbook (e.g., listing the top-k most reused patterns), and adjusting parameters of the induction (such as toggling certain heuristics, or specifying a new pattern through CELM-lang to guide the induction). This is critical for debugging and tuning, given the complex behavior of the grammar learning process. It also allows an operator or researcher to inject domain knowledge (for example, one could predefine a token structure for known file formats or known patterns, serving as a hint to the system).

A6. Eigentoken Mock Rules and Testing Suite: For development and demonstration, the system includes a set of **mock rules** – essentially hard-coded grammar rules for simple patterns – and a testing harness to verify correctness. These help ensure that as new features are added to the grammar induction engine or CELM-lang, basic invariants (like lossless reconstruction) remain intact. The mock rules also serve as examples for how grammar rules can be structured.

(A7. Roadmap: Replication/Erasure and LLM Integration) Beyond the scope of the initial project, we note some planned extensions: adding replication and erasure coding at the storage layer (applied on grammar leaves and/or token payload segments, to provide fault tolerance by duplicating or encoding tokens across multiple machines or data centers), and exploring how compiled ELMs might integrate with or complement traditional learned models (e.g., by providing architecture blueprints or pre-trained components).

4.2 Workflow and Data Ingestion

The workflow of the Eigentokens system during data ingestion and model compilation involves several stages:

1. **Data Ingestion:** New data (e.g., a file or object) is written to Eigentokens via the storage interface. The system performs Stage 1 ingestion (as described in A3): content-defined chunking and immediate token creation. The data is now stored (in duplicate-friendly form) and the client's write is acknowledged.

2. **Background Processing:** The new data's chunks are queued for analysis. The Grammar Induction Engine gradually processes these chunks (potentially along with others in a batch), discovering new patterns and creating higher-level tokens. The B^+-forest is updated with these new tokens. If certain raw chunks become redundant (fully represented by grammar rules), they may be marked for lazy compression or removal.

3. **Index Maintenance:** Periodically, the system triggers Stage 3 compaction on the B^+-forest. This might merge nearby tokens, compress literal data that can now be compressed, remove obsolete tokens, and rebalance trees for optimal lookup performance. A snapshot is taken so that, in event of failure, recovery can skip redoing already-applied grammar rules.

4. **Model Compilation:** At any point (on demand or at set intervals), the system can compile a deterministic language model (ELM) from the grammar tokens. This involves translating the grammar cookbook into a neural network structure and weights. In practice, this might be implemented by generating code or data that initializes an LLM's parameters according to the grammar rules. The details of this compilation are part of Section 5.

5. **Query and Evolution:** Users or processes can query the grammar (via the Analysis API) to understand the current patterns or manually inject new grammar rules (for instance, if an expert recognizes a pattern that the system hasn't yet, they could add it). This process effectively evolves the model and the storage format in tandem, under human guidance.

Overall, the system continuously cycles through ingestion and analysis. Data is always quickly stored and accessible, but the internal representation is continuously improving as more patterns are discovered. This "learn-as-you-store" paradigm ensures that storage efficiency and the implicit model quality both improve over time.

5 The Comdare Database as Eigentokens Substrate

The preceding chapters presented Eigentokens as an abstract architecture—a grammar-based storage system that simultaneously serves as a deterministic compiler for language models. This chapter grounds that architecture in the concrete system that implements it: the **Comdare database platform** (BEP Venture UG), a C++23 codebase of approximately 280,000 lines of code. We explain *why* this platform exists, *how* it enables the Eigentokens vision, and *what* emerges when storage infrastructure and grammar-based AI construction converge.

5.1 Why Build a Custom Database?

The Eigentokens concept requires capabilities that no existing system provides in combination:

1. **Deterministic arithmetic across platforms.** Every Eigentoken is identified by a SHA-512 hash, and the B^+-forest index performs arithmetic on these 512-bit values. Conventional databases rely on platform-native integer types, whose overflow and rounding behavior varies between x86_64, ARM64, and RISC-V. Eigentokens demand *bit-identical* results on all platforms—a precondition for deterministic model compilation. The Comdare platform addresses this with a custom **UBigInteger** type that serves as the universal processing element across all database subsystems: filters, fingerprints, deduplication strategies, cluster management, replication, and the object store. Every data operation flows through UBigInteger, ensuring that results are reproducible regardless of hardware.

2. **Grammar-aware storage boundaries.** Standard content-defined chunking (CDC) treats data as an undifferentiated byte stream. Eigentokens need chunk boundaries that *respect grammar structure*—splitting at syntactically meaningful positions rather than at arbitrary hash-triggered breakpoints. This requires deep integration between the storage engine and the pattern analysis layer, which is impractical to retrofit into existing databases or object stores.

3. **Unified storage and compilation.** Existing databases optimize for either storage efficiency or query performance, not for compiling AI models from the stored patterns. Eigentokens require the storage system to simultaneously maintain a queryable B^+-forest index *and* a "grammar cookbook" of production rules that can be translated into neural network weights. This dual purpose demands custom data structures and a processing pipeline that no off-the-shelf database provides.

4. **Asynchronous, multi-stage analysis.** The $O(n^3)$ grammar induction algorithm is too expensive to run synchronously on every write. The system must accept data at low latency ($<$10 ms per chunk) while performing deep structural analysis in the background. This requires a purpose-built asynchronous pipeline with careful coordination between immediate storage, background pattern discovery, and index compaction.

In short: the Eigentokens concept treats the storage system as an AI system in its own right. Building on top of an existing database would force compromises that undermine the core thesis—that storage deduplication and model construction are the *same operation* viewed from different angles.

5.2 Architectural Means: How the Platform Is Built

The Comdare platform is organized around a hierarchical module system called the **Baugruppen Principle** (assembly principle):

Bauteile (Components) are atomic modules with no internal dependencies—individual algorithms or data structures (e.g., a single fingerprint algorithm, a single compression codec). Each lives in its own Git repository.

Baugruppen (Assemblies) compose Bauteile into functional subsystems. They are denoted by the `-all` suffix (e.g., `comdare-foundation-all`) and define baseline tiers of integrated functionality.

Produkte (Products) combine multiple Baugruppen into user-facing applications. `comdare-db` is the Eigentokens product, consuming 23 Baugruppen.

The platform provides four foundational capabilities that directly enable the seven Eigentokens components (A1–A7):

Arbitrary-Precision Arithmetic. The foundation layer implements BigInteger multiplication at three complexity levels—schoolbook $O(n^2)$, Karatsuba $O(n^{1.585})$ [KO62], and NTT-based $O(n \log n)$—ensuring that SHA-512 hash arithmetic (comparisons, range queries, frequency counting) scales efficiently. Montgomery modular arithmetic supports the number-theoretic operations needed for grammar rule hashing.

High-Throughput I/O and Concurrency. Linux `io_uring` [Axb19] provides zero-copy asynchronous I/O for the three-stage ingestion pipeline (A3). Lock-free data structures, Conflict-free Replicated Data Types (CRDTs) [SPBZ11], and consistent hash rings enable concurrent grammar induction across multiple threads and—eventually—distributed nodes, without sacrificing determinism.

Content-Addressed Storage and Compression. The storage layer provides SHA-512-based content addressing, seekable compression integration (zstd-seekable, BGZF [Li11], LZMA), and HTTP Range support per RFC 9110 [FNR22]. These capabilities directly implement the B^+-forest (A2) and Storage Interface (A4) components, including token-aligned block maps that enable random access to grammatically-compressed data.

Multi-Platform Build Infrastructure. A custom build system (1,921 features, 21 modules) compiles and tests every component on 12 runners spanning x86_64, ARM64, macOS (Intel and Apple Silicon), Raspberry Pi, and RISC-V. This multi-platform validation is not merely a convenience—it is the mechanism that *verifies* the determinism guarantee: if the UBigInteger-based processing produces identical results on all six architectures, the Eigentokens compilation is provably platform-independent.

5.3 What Emerges: The Database as Learning System

When the Comdare database is combined with the Eigentokens architecture described in Chapters 3–4, a fundamentally new kind of system emerges:

5.3.1 Storage That Discovers Knowledge

Traditional databases store data passively. The Comdare/Eigentokens system *analyzes* every byte it stores. As data enters through the S3/KV interface (A4), the asynchronous pipeline (A3) performs two operations simultaneously:

1. **Immediate storage:** Data is chunked, hashed, and written to the B^+-forest (A2) with sub-10 ms latency. The data is immediately accessible.
2. **Background learning:** The Grammar Induction Engine (A1) scans the new data against the global corpus, discovering repeated patterns, creating new Eigentokens (production rules), and enriching the grammar cookbook. Over time, the internal representation of the data becomes increasingly compressed and structured.

The result is a storage system whose efficiency *improves the more data it processes*—not because of tuning, but because it learns the structure of its contents. The target is 25–40% better deduplication than FastCDC, achieved through cross-object grammar discovery rather than byte-level hashing.

5.3.2 Grammar as a Compilation Blueprint

The grammar cookbook accumulated by the storage engine is not merely a compression artifact. Each production rule captures a structural pattern in the data—a piece of "knowledge" that can be reused. The Eigentokens system interprets these rules as a blueprint for constructing a **deterministic language model (ELM)**:

- Production frequencies determine neural network weights (no gradient descent required).
- The hierarchical token structure defines the network architecture (no hyperparameter search).
- Topic-filtered B^+-trees enable *domain-specific* model compilation: a model for code, a model for scientific text, a model for structured data—all from the same storage engine, by selecting different grammar cookbook subsets.

The compiled model is 100% deterministic: identical input data always produces an identical model with identical outputs. This eliminates the floating-point nondeterminism that makes conventional LLM training irreproducible, and enables *complete traceability*—every model weight can be traced back to specific source data patterns.

5.3.3 Incremental Evolution Instead of Retraining

Because the grammar is maintained as a living data structure within the database, new knowledge can be integrated without retraining:

- **Adding data:** New patterns are discovered and added to the cookbook. Affected model components can be recompiled incrementally.
- **Correcting errors:** A faulty grammar rule can be identified (via the Analysis API, A5), removed or modified, and the model recompiled with the corrected rule. The fix propagates deterministically.
- **Specializing:** An operator can inject domain-specific rules via CELM-lang (Section 3.3) to guide the grammar induction toward patterns relevant to a particular application.

This stands in sharp contrast to conventional LLMs, where correcting a single factual error typically requires fine-tuning or retraining on corrected data—an expensive, probabilistic process with unpredictable side effects.

5.3.4 The Omni-LLM Vision

At the highest level of abstraction, the combined system points toward what we call the **Omni-LLM**: a universal language model that operates by executing stored grammar rules rather than sampling from probability distributions. In this vision:

- The **storage layer** (Comdare) continuously ingests and analyzes data from all domains.
- The **grammar cookbook** grows into a comprehensive knowledge base spanning code, text, structured data, and multimedia.
- **Model compilation** produces specialized or general-purpose models on demand, each fully deterministic and traceable.

- **CELM-lang** (operating at the M2/M3 metamodel levels) enables the system to analyze and improve its own grammar—a form of *agentic self-improvement* grounded in explicit rules rather than opaque weight updates.

Achieving parity with neural LLMs in open-ended generation remains a distant goal. However, for domains where *reliability, traceability, and determinism* are more valuable than raw fluency—such as legal analysis, medical documentation, financial compliance, and safety-critical systems—the grammar-based approach offers guarantees that probabilistic models fundamentally cannot provide.

5.4 Current Status and Scale

The Comdare platform comprises approximately 280,000 lines of C++23 code across 250+ repositories, with a weighted implementation status of 88%. Nine of the 19 major Baugruppen exceed 90% completion. The database product (`comdare-db`) organizes 65 modules into seven baseline tiers totaling 75,685 lines of verified source code.

The CELM-specific components (A1–A7) are fully specified at the architectural level—documented in a comprehensive research expose with 43 references, 2 core research questions, and 13 identified novel contributions—and await C++ implementation using the production infrastructure described above.

Prior validation of the grammar-aware deduplication approach was performed in two production scenarios: a self-hosting experiment on 380,000 lines of C++ code (demonstrating cross-object pattern discovery) and an industrial deployment on 1 petabyte of video data (Bosch XC Abstatt, demonstrating petabyte-scale viability). These results establish confidence that the full Eigentokens system—including deterministic model compilation, which the earlier prototype lacked—can operate at the target scale.

6 Implementation Architecture: Technical Detail

This chapter provides a detailed technical inventory of the Comdare platform's module hierarchy, baseline tiers, foundation infrastructure, and build system. It complements the preceding chapter's goal-oriented narrative with concrete metrics and structural mappings.

6.1 Module Hierarchy: The Baugruppen Principle

The implementation follows a strict hierarchical composition model with three levels of abstraction:

Bauteil (Component): An atomic, self-contained module with no internal dependencies. Examples include the SIMD vectorization library or individual fingerprint algorithm implementations (e.g., `comdare-fingerprint-sha`, `comdare-fingerprint-fnv1a`). Each Bauteil resides in its own Git repository and is individually versioned, enabling fine-grained dependency management and independent release cycles.

Baugruppe (Assembly): A composition of Bauteile, denoted by the `-all` suffix (e.g., `comdare-foundation-all`). Baugruppen define baseline tiers and provide integrated functionality across their constituent components. They serve as the primary dependency units for products. Each Baugruppe maintains its own baseline hierarchy, integration tests, and documentation.

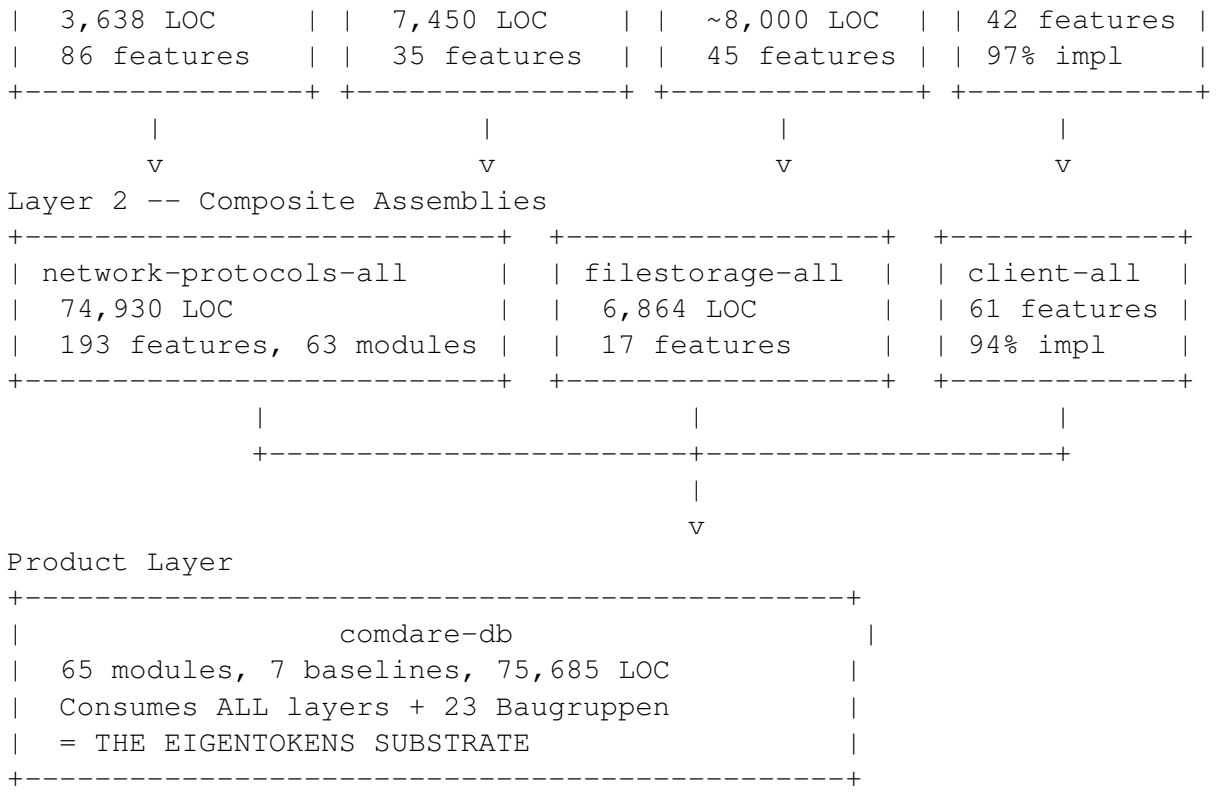
Produkt (Product): A user-facing application built atop multiple Baugruppen, providing defined interfaces and deployment artifacts. The database system (`comdare-db`) is the primary product relevant to Eigentokens, consuming 23 external Baugruppe dependencies.

This three-tier model ensures that each module can be independently developed, tested, and versioned while maintaining clear dependency boundaries. Placeholder modules serve as procurement notes for planned but not yet implemented functionality, preserving the intended architecture even before code exists.

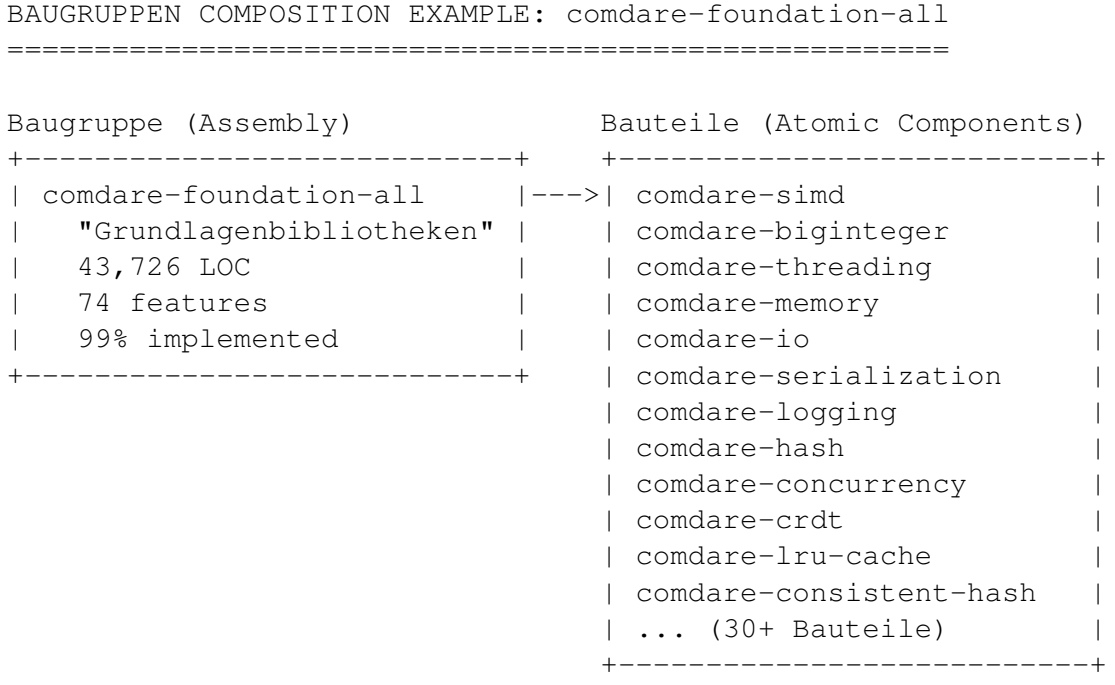
6.1.1 Module Dependency Graph

The composition follows a directed acyclic graph (DAG) structure where higher-tier modules depend on lower-tier modules but never the reverse. The following diagram shows the complete dependency structure:

```
DEPENDENCY LAYERS (Directed Acyclic Graph, bottom-up)
=====================================================

Layer 0 -- Roots (no dependencies)
+-----------------+  +--------------+  +--------------+
| foundation-all  |  |  config-all  |  | external-all |
|  43,726 LOC     |  |  5,014 LOC   |  |  3rd-party   |
|  74 features    |  |  35 features |  |  wrappers     |
+-----------------+  +--------------+  +--------------+
        |                   |                 |
        v                   v                 v
Layer 1 -- Foundation Dependents
+----------------+ +---------------+ +--------------+ +-------------+
| encryption-all | | licensing-all | | archiving-all| | storage-all |
```



Concrete examples illustrate the three-tier composition:



6.2 Baseline Tier Architecture

The `comdare-db` product organizes its 65 internal modules into seven baseline tiers (B0–B6), each building upon the previous:

The B2 tier deserves special attention as it implements the **filter engine** (33 headers, 21,229 LOC)—the largest single module—which provides the pattern-matching infrastructure that the Grammar Induction

Table 6.1: comdare-db Baseline Tier Architecture

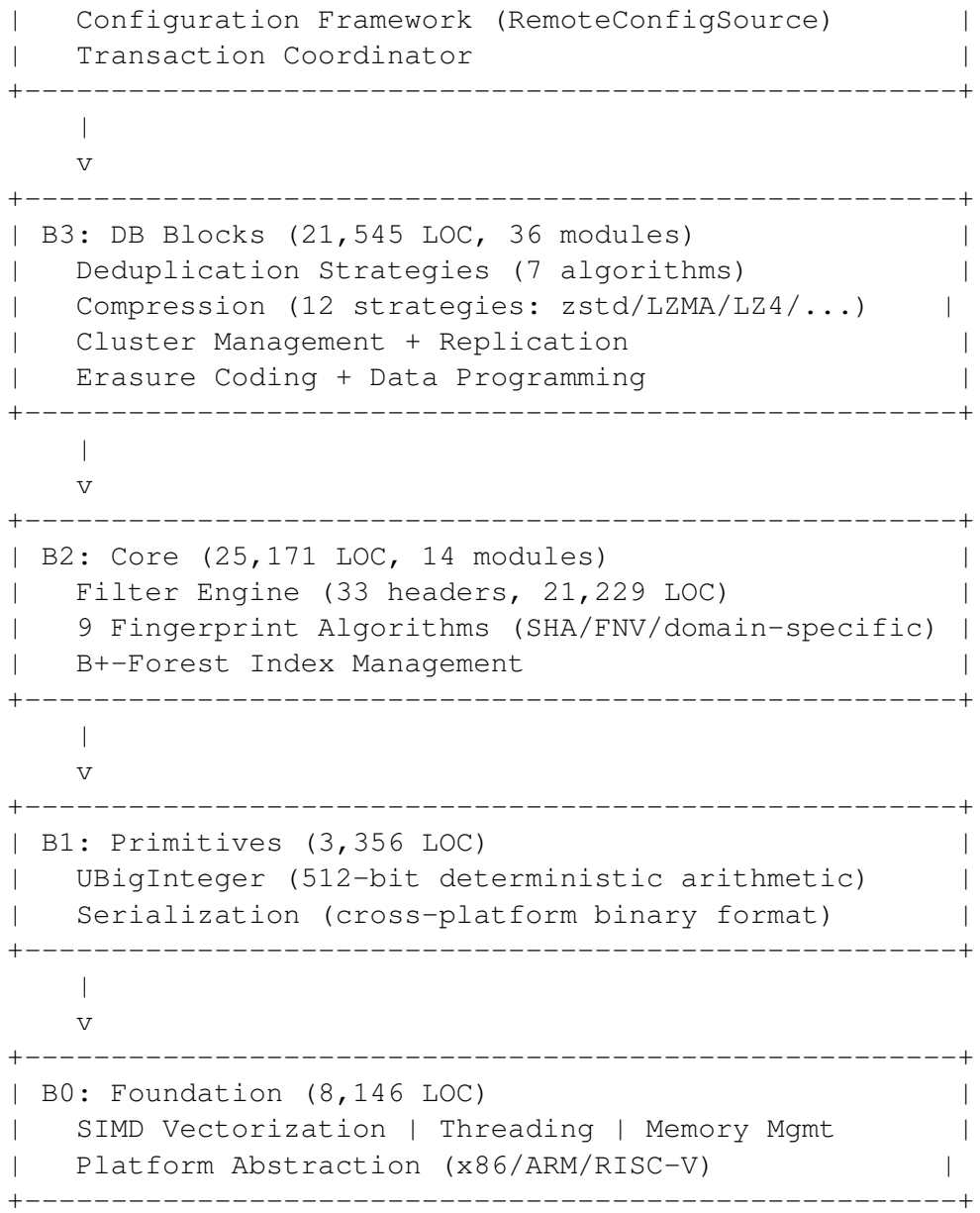
Tier	Domain	Modules	LOC	Key Responsibilities
B0	Foundation	5	8,146	Concurrency, memory management, SIMD, threading
B1	Primitives	2	3,356	BigInteger arithmetic, serialization
B2	Core	14	25,171	Filter engine, 9 fingerprint algorithms, index management
B3	DB Blocks	36	21,545	Cluster, deduplication, replication, 12 compression strategies
B4	DB Core	4	15,306	DatabaseCore orchestrator, configuration framework
B5	Interfaces	3	1,195	Object storage API, PostgreSQL HA client
B6	User Interface	1	966	CLI, Web, Qt, Node.js interface stubs
	Total	**65**	**75,685**	

Engine (A1) builds upon. The nine fingerprint algorithms in B2 (SHA-256, SHA-512, FNV-1a, and six domain-specific variants) enable multi-level content-addressable indexing, where different hash functions serve different roles: SHA-512 for global deduplication keys, FNV-1a for fast similarity detection, and domain-specific hashes for topic-filtered B^+-forest trees.
The B3 tier implements 12 compression strategies that the Asynchronous Pipeline (A3) can select dynamically based on data characteristics:

- **Block-level:** zstd (levels 1–22), LZMA, LZ4, Snappy
- **Seekable:** zstd-seekable with configurable frame sizes, BGZF [Li11]
- **Grammar-aware:** Token-aligned compression where boundaries respect Eigentoken structure
- **Domain-specific:** Columnar encoding for tabular data, delta encoding for versioned objects

The following diagram shows how data flows upward through the baseline tiers:

```
comdare-db BASELINE TIER STACK
==============================

User Request (S3 PUT / KV Write / SQL INSERT)
    |
    v
+-----------------------------------------------------+
| B6: User Interface (966 LOC)                        |
|   CLI / Web / Qt / Node.js                          |
+-----------------------------------------------------+
    |
    v
+-----------------------------------------------------+
| B5: Interfaces (1,195 LOC)                          |
|   S3-compatible Object Storage API                  |
|   PostgreSQL HA Client (Wire Protocol)              |
+-----------------------------------------------------+
    |
    v
+-----------------------------------------------------+
| B4: DB Core (15,306 LOC)                            |
|   DatabaseCore Orchestrator                         |
```



6.3 Foundation Infrastructure

The Eigentokens system draws on several major Baugruppen that provide cross-cutting infrastructure. These modules represent the bulk of the system's 280,000+ lines of code and are at or near production quality.

6.3.1 comdare-foundation-all (43,726 LOC, 74 features, 99% implemented)

The absolute foundation layer provides:

- **BigInteger Arithmetic (∼8,000 LOC):** Multiple multiplication algorithms—schoolbook $O(n^2)$, Karatsuba $O(n^{1.585})$ [KO62], and Number Theoretic Transform (NTT)-based $O(n \log n)$ multiplication using a two-prime Chinese Remainder Theorem (CRT) scheme. Montgomery modular arithmetic for efficient modular exponentiation. Miller–Rabin primality testing. Optional GMP interop layer with conditional compilation for environments where GNU Multiple Precision Arithmetic Library is available.

- **Concurrency Primitives (∼6,000 LOC):** Lock-free data structures, thread-local algorithm tracing (introspection), consistent hash rings for distributed token placement, five Conflict-free Replicated Data Type (CRDT) implementations [SPBZ11] (G-Counter, PN-Counter, G-Set, OR-Set, LWW-Register with StorageState aggregate), and $O(1)$ hash-plus-list LRU cache templates.
- **I/O Scheduling (∼4,500 LOC):** Linux `io_uring` integration [Axb19] for zero-copy asynchronous I/O, writer-preference single-writer/multiple-reader (SWMR) concurrency, batch I/O operations, and machine-learning-based compression heuristics that select optimal compression strategies at runtime.
- **Platform Abstraction (∼5,000 LOC):** Hardware detection for x86_64, ARM64 (including Apple Silicon), and RISC-V; SIMD vectorization library with compile-time dispatch; memory management with custom allocators; and serialization frameworks for cross-platform data exchange.

The BigInteger library is particularly critical: since all Eigentokens are identified by SHA-512 hashes (512-bit integers), the system requires efficient arbitrary-precision arithmetic for hash comparison, range queries over the B^+-forest, and the frequency-based weight calculation used in deterministic model compilation. The NTT-based $O(n \log n)$ multiplier enables these operations to scale to hash spaces that would be impractical with naive algorithms.

6.3.2 comdare-network-protocols-all (74,930 LOC, 193 features, 95% implemented)

The largest Baugruppe by code volume, organized into seven internal baselines with 63 sub-modules:

- **Baseline 0 (Foundation, 12 modules):** Threading primitives, IP-layer basics, connection retry/proxy/session management
- **Baseline 1 (Primitives, 8 modules):** Symmetric and asymmetric protocol support, multicast, custom UDP and TCP implementations
- **Baseline 2 (Core, 10 modules):** Latency optimization, throughput balancing, multi-stream coordination
- **Baseline 3 (Streaming, 12 modules):** Tree-based error recovery, frame reassembly for large token transfers
- **Baseline 4 (Interfaces, 15 modules):** Package flow management, auto-balancing, fingerprint-based routing, hardware offloading
- **Baseline 5 (Admin, 4 modules):** Shell, Qt, Web, and binary admin pipe interfaces
- **Baseline 6 (UI, 2 modules):** User-facing protocol dashboards

The admin pipe protocol is notable: it provides an ABI-stable binary plugin system with 45+ protocol repositories, enabling runtime extension of the storage interface (A4) without recompilation. This mechanism is used by the Analysis API (A5) to expose grammar metrics and by CELM-lang to inject new interpretation program types at runtime.

6.3.3 comdare-encryption-all (3,638 LOC, 86 features, 94% implemented)

A hierarchical strategy-pattern encryption layer:

- **Symmetric:** AES-256-GCM, XChaCha20-Poly1305
- **Asymmetric:** RSA-4096, ECDH with curves P-256 and P-384

- **TLS Integration (981 LOC):** Custom TLS infrastructure with certificate pinning (209 LOC) and forward secrecy through ephemeral key generation
- **Combined Strategy (277 LOC):** Decorator and visitor patterns for composing hybrid encryption pipelines

Encryption is relevant to Eigentokens in two ways: (1) tokens containing sensitive data can be encrypted before storage while preserving the grammar structure (encryption is applied at the payload level, not the token level), and (2) the secure token transport protocol ensures that grammar rules transmitted between distributed nodes cannot be tampered with.

6.3.4 comdare-storage-all (42 features, 97% implemented)

Generic storage abstractions that the database product specializes:

- Content-addressed storage with SHA-512 hashing for Eigentoken identification
- Seekable compression integration (BGZF, zstd-seekable, LZMA) with $<5\%$ overhead on range reads
- HTTP Range support per RFC 9110 [FNR22] with token-aligned block maps
- Write-ahead logging and periodic snapshots for crash consistency

Together, these four Baugruppen account for over 120,000 lines of code and provide the runtime substrate on which grammar induction, B^+-forest indexing, and deterministic model compilation operate.

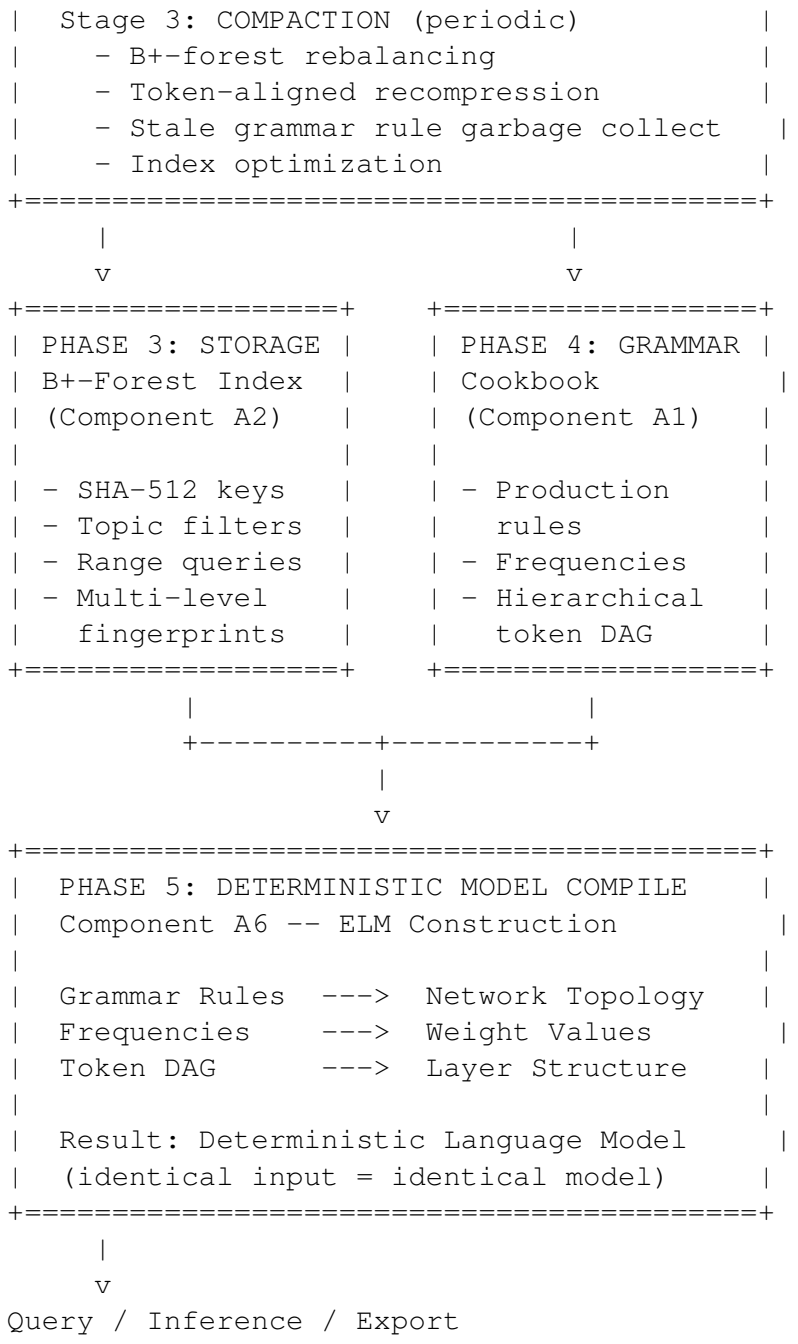
7 System Pipelines and the Role of the Build System

This chapter presents the dynamic behavior of the Eigentokens/Comdare system through flow diagrams and pipeline descriptions. It also explains the critical role that the custom build system plays in enabling a 280,000-line multi-platform C++ codebase to function as a coherent research platform.

7.1 End-to-End Data Flow: From Ingestion to Model Compilation

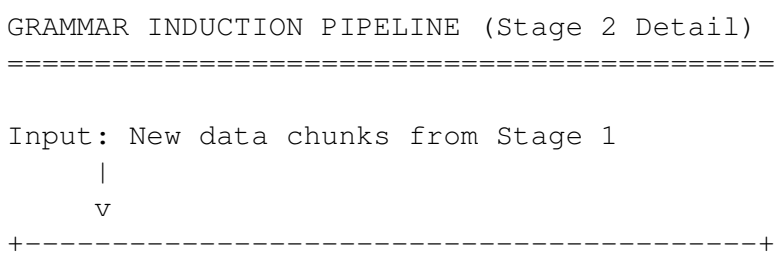
The complete lifecycle of data through the Eigentokens system spans five phases. The following diagram shows the end-to-end flow from data ingestion through grammar discovery to deterministic model compilation:

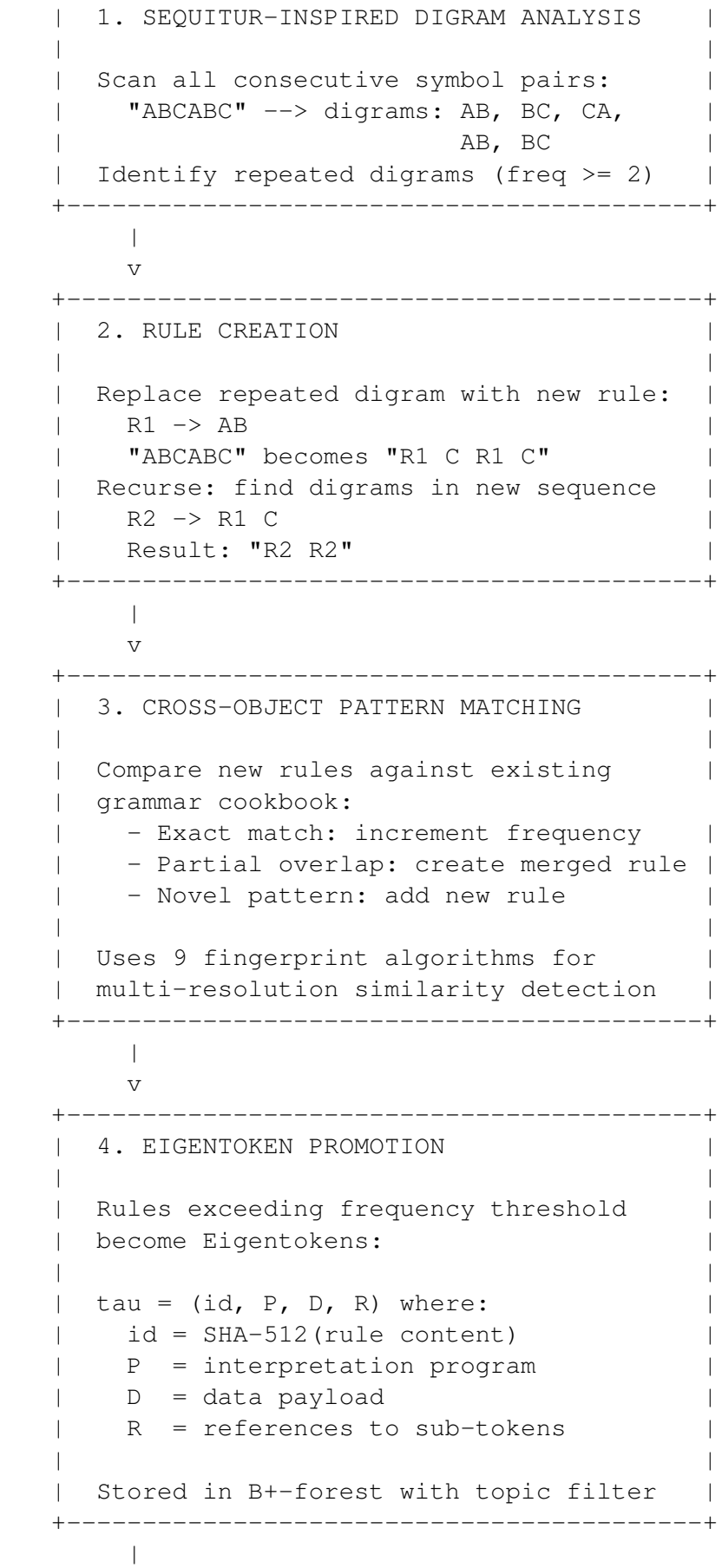
```
EIGENTOKENS END-TO-END DATA FLOW
================================

External Data Sources
(files, streams, API uploads)
     |
     v
+=========================================+
|  PHASE 1: INGESTION (S3/KV Interface)    |
|  Component A4 -- Storage Interface        |
|                                          |
|  HTTP PUT/POST  --->  Authentication     |
|  S3 Multipart   --->  Rate Limiting      |
|  KV Write       --->  Request Routing    |
+=========================================+
     |
     v
+=========================================+
|  PHASE 2: THREE-STAGE ASYNC PIPELINE     |
|  Component A3 -- Asynchronous Processing  |
|                                          |
|  Stage 1: IMMEDIATE (<10ms)              |
|    - Content-defined chunking            |
|    - SHA-512 hash computation            |
|    - Dedup check against B+-forest       |
|    - Write to storage (WAL + data)       |
|    - Return acknowledgment to client     |
|                                          |
|  Stage 2: BACKGROUND (seconds-minutes)   |
|    - Grammar induction O(n^3)            |
|    - Cross-object pattern discovery      |
|    - Eigentoken creation/refinement      |
|    - Grammar cookbook update              |
|    - Frequency statistics maintenance    |
|                                          |
```



7.2 Grammar Induction Pipeline (Stage 2 Detail)

The grammar induction process—the intellectual core of the Eigentokens concept—transforms raw byte sequences into structured grammar rules. This is the $O(n^3)$ algorithm that runs asynchronously in Stage 2:



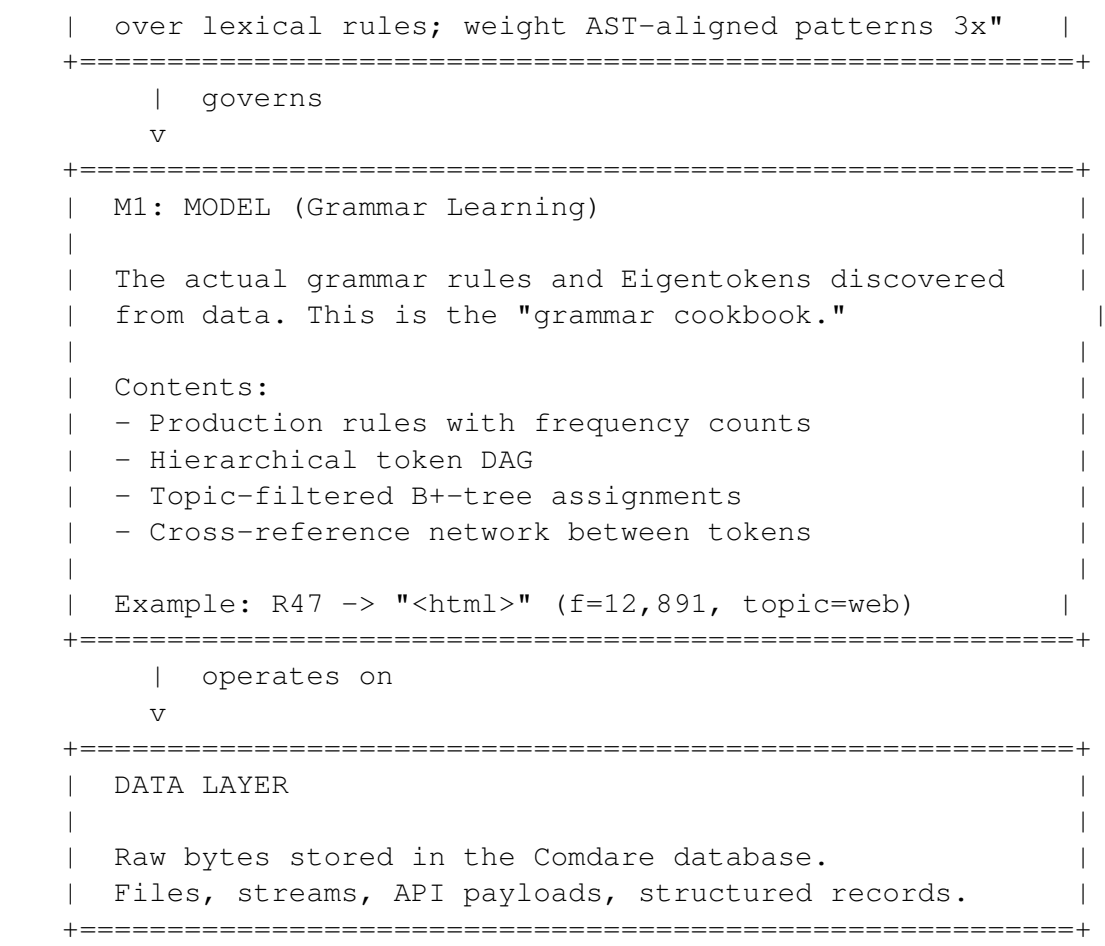

1. SEQUITUR-INSPIRED DIGRAM ANALYSIS
Scan all consecutive symbol pairs:
"ABCABC" --> digrams: AB, BC, CA,
AB, BC
Identify repeated digrams (freq >= 2)
2. RULE CREATION
Replace repeated digram with new rule:
R1 -> AB
"ABCABC" becomes "R1 C R1 C"
Recurse: find digrams in new sequence
R2 -> R1 C
Result: "R2 R2"
3. CROSS-OBJECT PATTERN MATCHING
Compare new rules against existing
grammar cookbook:
- Exact match: increment frequency
- Partial overlap: create merged rule
- Novel pattern: add new rule
Uses 9 fingerprint algorithms for
multi-resolution similarity detection
4. EIGENTOKEN PROMOTION
Rules exceeding frequency threshold
become Eigentokens:
tau = (id, P, D, R) where:
id = SHA-512(rule content)
P = interpretation program
D = data payload
R = references to sub-tokens
Stored in B+-forest with topic filter

```
+-----------------------------------------+
|  5. GRAMMAR COOKBOOK UPDATE              |
|                                         |
|  +-- Production Rules ------+           |
|  | R1 -> "http://"    (f=892)|          |
|  | R2 -> R1 "www."   (f=445)|          |
|  | R3 -> "<div"      (f=2031)|         |
|  | R4 -> R3 " class" (f=967)|          |
|  | ...                      |          |
|  +--------------------------+          |
|                                         |
|  Frequencies directly map to neural     |
|  network weights in Phase 5             |
+-----------------------------------------+
```

7.3 CELM-lang Metamodel Architecture

The CELM-lang specification operates at three metamodel levels, each governing a different aspect of the grammar-based system. This hierarchy enables the system to not only process data but also analyze and improve its own processing rules:

```
CELM-LANG METAMODEL HIERARCHY
==============================

+============================================================+
|  M3: META-METAMODEL (Self-Adaptation)                      |
|                                                            |
|  Defines rules for modifying M2 rules.                     |
|  Enables: Agentic self-improvement of the grammar          |
|  system. The database can discover that certain M2         |
|  strategies are suboptimal and automatically adjust        |
|  them based on observed data patterns.                     |
|                                                            |
|  Example: "If compression ratio drops below threshold,     |
|  switch from digram-based to n-gram-based induction"       |
+============================================================+
     |  governs
     v
+============================================================+
|  M2: METAMODEL (Grammar Application)                       |
|                                                            |
|  Defines how grammar rules are applied, composed,          |
|  and optimized. Controls the behavior of M1.               |
|                                                            |
|  Components:                                               |
|  - Rule composition strategies (merge, split, refine)      |
|  - Topic filter configuration                              |
|  - Compression/dedup trade-off policies                    |
|  - Model compilation parameters                            |
|                                                            |
|  Example: "For source code, prefer structural rules        |
```



The key insight is that M2 and M3 are themselves stored as Eigentokens within the database. The system's self-improvement rules are subject to the same deterministic, traceable processing as user data. This means every adaptation decision can be audited: "the system switched from strategy X to strategy Y because rule R at M3 level triggered, based on metrics M."

7.4 Comparative Landscape

The following table positions the Eigentokens/Comdare approach against existing systems across six critical dimensions. No existing system addresses more than two dimensions simultaneously:

Grammar-Aware: Chunk boundaries respect syntactic structure rather than byte-level hashing.

Deterministic: Bit-identical results across all supported hardware platforms.

Cross-Object: Pattern discovery spans multiple objects and data domains.

Model Compile: Storage patterns can be compiled into neural network weights.

Multi-Platform: Verified operation on x86_64, ARM64, RISC-V, and macOS.

Incremental: New data can be integrated without reprocessing the entire corpus.

Table 7.1: Eigentokens vs. Existing Approaches

System	Grammar-Aware	Deterministic	Cross-Object	Model Compile	Multi-Platform	Incremental
FastCDC	—	✓	—	—	✓	—
BGZF	—	✓	—	—	✓	—
RocksDB	—	✓	—	—	partial	✓
UltiHash	—	partial	✓	—	—	✓
DuckDB	—	✓	—	—	✓	—
Eigentokens	✓	✓	✓	✓	✓	✓

7.5 The Role of the Build System

7.5.1 Why a Custom Build System?

A 280,000-line C++23 codebase distributed across 250+ Git repositories on six hardware platforms cannot be managed with conventional build tools. The challenges are:

1. **The Chicken-and-Egg Problem.** The build system must compile projects, but the build system *itself* is a C++ project that needs to be compiled. Standard solutions (requiring a pre-installed build system) create a hard dependency on a specific host environment. The Comdare build system solves this with a multi-stage bootstrap:

```
BUILDSYSTEM BOOTSTRAP (Chicken-and-Egg Resolution)
==================================================

Stage 0: Bare Environment (only OS + shell)
     |
     v
+-----------------------------------------+
|  BOOTSTRAP PHASE                        |
|  cd-buildsystem-dependency-manager      |
|                                         |
|  1. Detect platform (OS, CPU, shell)    |
|  2. Download minimal toolchain:         |
|     - CMake (if not present)            |
|     - Ninja (if not present)            |
|     - GCC or Clang (if not present)     |
|  3. Compile cd-buildsystem-core         |
|  4. Core now compiles remaining modules |
+-----------------------------------------+
     |
     v
Full Build System Available
(can now compile any of the 250+ projects)
```

2. **Consistency Across 250+ Repositories.** Each project must build identically regardless of which developer machine or CI runner compiles it. The build system enforces this through XML-based

build declarations (`buildsystem.xml`) that replace per-project `CMakeLists.txt` files:

```
buildsystem.xml DECLARATION (per project)
=========================================

<buildsystem version="3.0">
  <project name="comdare-fingerprint-sha">
    <type>bauteil</type>
    <language>cpp23</language>
    <dependencies>
      <dependency>comdare-foundation-all</dependency>
    </dependencies>
    <build>
      <source-dir>src</source-dir>
      <include-dir>include</include-dir>
      <test-dir>tests</test-dir>
    </build>
    <dimensions>
      <d0>standard</d0>    <!-- base configuration -->
      <d3>zstd,lz4</d3>    <!-- compression backends -->
      <d7>avx2,neon</d7>   <!-- SIMD targets -->
    </dimensions>
  </project>
</buildsystem>

The build system reads this declaration and generates:
- CMake configuration (platform-specific)
- Dependency resolution (recursive download)
- Compiler flags (per-dimension)
- Test targets
- Artifact packaging
```

3. **Bridge-Pattern Module Delegation.** Each Layer 1 build system module implements its own `interface.sh` (Linux/macOS), `interface.bat` (Windows), and `interface.cmake` (cross-platform) delegation scripts. Build logic resides in the *responsible* module, not in a centralized parser:

```
BRIDGE PATTERN: BUILD LOGIC DELEGATION
======================================

cd-buildsystem-core (Orchestrator)
     |
     |--- "How do I validate?"
     |         |
     |         v
     |    cd-buildsystem-validation-system/
     |         interface.sh  (6-phase validation logic)
     |         interface.cmake
     |
     |--- "What compiler to use?"
     |         |
     |         v
     |    cd-buildsystem-compiler-manager/
```

```
|         interface.sh  (GCC/Clang/MSVC selection)
|         interface.cmake
|
|--- "How do I resolve dependencies?"
|         |
|         v
|    cd-buildsystem-dependency-manager/
|         interface.sh  (recursive download + bootstrap)
|         interface.cmake
|
|--- "What build mode?"
|         |
|         v
|    cd-buildsystem-build-mode-system/
|         interface.sh  (DEV/DEV_PLUS/PRODUCTION)
|         interface.cmake
|
+--- (9 L1 modules total, each with own interface)
```

4. **Multi-Platform Determinism Verification.** The build system's most critical role for Eigentokens is *proving* that results are platform-independent. It achieves this by compiling and running identical test suites on all target platforms:

7.5.2 The Six-Phase Build Pipeline

Every project passes through a six-phase validation pipeline. Phases 0–3 are mandatory for all builds; phases 4–5 are activated for performance-critical modules:

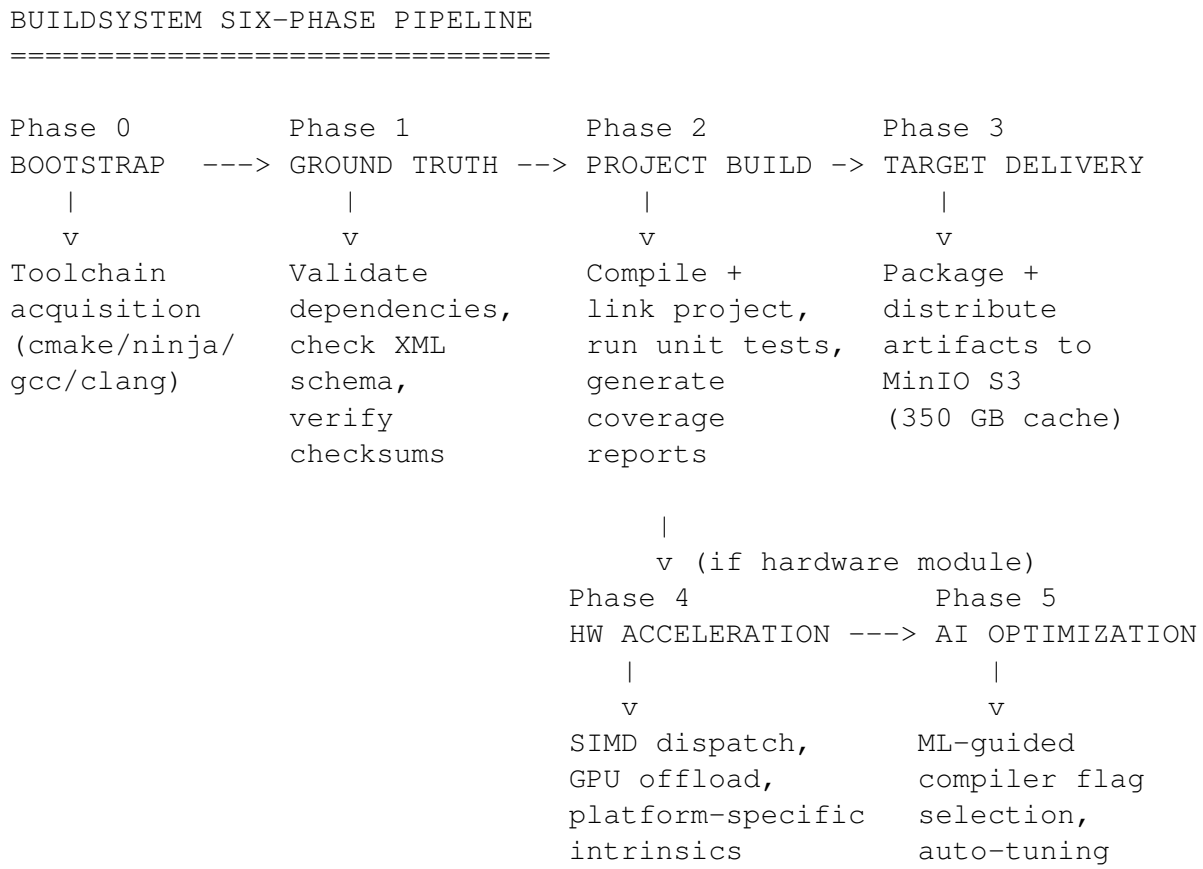


Table 7.2: Build System Six-Phase Pipeline Details

Phase	Name	Purpose	Status
0	Bootstrap	Toolchain acquisition from bare OS	Active
1	GroundTruth	Dependency validation, XML schema check	Active
2	ProjectBuild	Compile, link, unit test, coverage	Active
3	TargetDelivery	Artifact packaging, MinIO upload	Active
4	HardwareAcceleration	SIMD dispatch, GPU offload	Reserved
5	AIOptimization	ML-guided flag selection, auto-tuning	Reserved

7.5.3 Build Modes and Cookbooks

The build system supports three build modes that control optimization levels, debug symbols, and feature gating. Each mode is defined by a "cookbook"—a set of compiler flags, preprocessor definitions, and linker options:

```
BUILD MODES (Cookbook System)
============================

+-------------------+-------------------------------------------+
| DEV               | -O0 -g3 -fsanitize=address,undefined      |
|                   | All assertions enabled, verbose logging   |
|                   | Used for: Local development, debugging    |
+-------------------+-------------------------------------------+
| DEV_PLUS          | -O2 -g1 -DNDEBUG=0                        |
|                   | Partial optimization, some assertions     |
|                   | Used for: CI pipelines, integration tests |
+-------------------+-------------------------------------------+
| PRODUCTION        | -O3 -march=native -flto -DNDEBUG          |
|                   | Full optimization, no assertions          |
|                   | Used for: Release builds, benchmarks      |
+-------------------+-------------------------------------------+

Dimension System (D0-D9):
D0: Base configuration (always active)
D1: Debug level
D2: Sanitizer selection
D3: Compression backends (zstd, lz4, lzma, snappy)
D4: Encryption backends (openssl, boringssl)
D5: Network protocol selection
D6: Database engine features
D7: SIMD targets (SSE4.2, AVX2, NEON, SVE)
D8: Platform-specific optimizations
D9: Experimental features
```

The dimension system (D0–D9) enables combinatorial configuration: a single project can be built with different compression backends, encryption libraries, and SIMD targets by specifying the desired dimensions in `buildsystem.xml`. This is essential for the multi-platform Eigentokens evaluation, where the same deduplication algorithm must be tested with different hardware acceleration paths.

7.5.4 How the Build System Serves Eigentokens

The build system is not merely a development convenience—it is a *scientific instrument* for the Eigentokens research. Its specific contributions to the research goals are:

1. **Determinism Proof (RQ1).** The 12-runner CI/CD architecture compiles and runs every test on six hardware platforms. If the UBigInteger-based grammar induction produces identical results on all platforms, the determinism claim is empirically validated. The build system automates this cross-platform verification on every commit.

2. **Performance Benchmarking (RQ2, RQ3).** The PRODUCTION build mode with `-O3 -march=native -flto` ensures that benchmark results reflect optimized code. The artifact system stores benchmark results per platform/architecture/build_ID in MinIO, enabling systematic comparison across hardware.

3. **Reproducibility (RQ5).** The XML-based build declaration ensures that any researcher can reproduce the exact build environment. The bootstrap phase downloads specific compiler versions (GCC 14.2, Clang 18.1) rather than relying on system-installed compilers, eliminating "works on my machine" problems.

4. **Scale Management (all RQs).** Managing 250+ repositories with consistent build configurations would be intractable without the bridge-pattern delegation. The build system enables the research team to focus on algorithm development rather than build infrastructure.

7.6 CI/CD Architecture: 12-Runner Multi-Platform Validation

The continuous integration infrastructure validates every code change across all target platforms. The 12 runners are organized into three tiers of reliability:

```
CI/CD 12-RUNNER ARCHITECTURE
============================

TIER 1: MANDATORY (build must pass on all)
+-------------------------------------------------+
|  Kubernetes Runners (4x Docker containers)      |
|                                                 |
|  +----------+  +----------+  +----------+  +--+ |
|  | Runner 6a|  | Runner 6b|  | Runner 6c|  |6d| |
|  | Talos    |  | Talos    |  | Talos    |  |  | |
|  | Node 1   |  | Node 2   |  | Node 3   |  |N4| |
|  +----------+  +----------+  +----------+  +--+ |
|  Tag: docker    Image: gcc:14 / clang:18        |
|  OS: Linux      Arch: x86_64 (container)        |
+-------------------------------------------------+
|  Bare-Metal x86_64 Servers (4x Shell runners)   |
|                                                 |
|  +-------+  +-------+  +-------+  +-------+     |
|  | pve1  |  | pve2  |  | node3 |  | node4 |     |
|  |Debian |  |Debian |  |Ubuntu |  |Ubuntu |     |
|  |ID: 11 |  |ID: 12 |  |ID: 13 |  |ID: 14 |     |
|  +-------+  +-------+  +-------+  +-------+     |
|  Tag: shell,x86_64    Native compilers          |
+-------------------------------------------------+
```

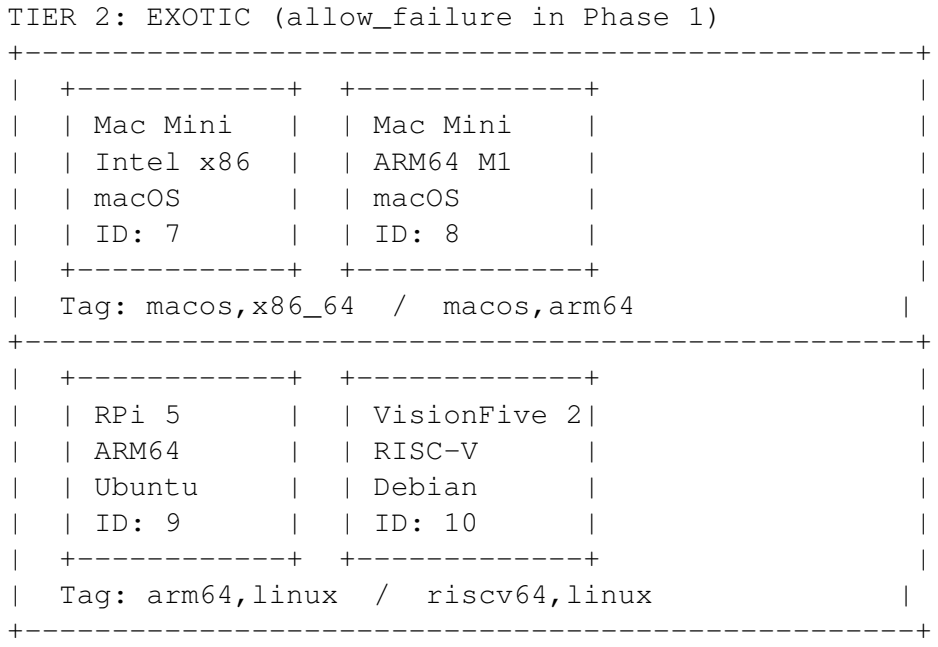
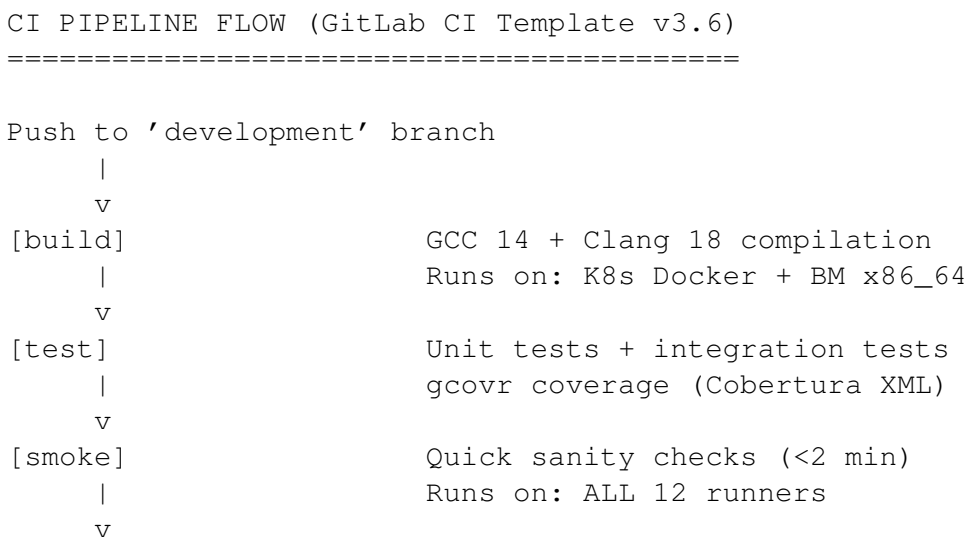


Table 7.3: CI/CD Runner Inventory

ID	Host	OS	Arch	Type	Tier
6a–d	K8s Pods	Linux (container)	x86_64	Docker	Mandatory
7	Mac Mini	macOS 14	x86_64	Shell	Exotic
8	Mac Mini	macOS 14	ARM64	Shell	Exotic
9	RPi 5	Ubuntu 24.04	ARM64	Shell	Exotic
10	VisionFive	Debian 13	RISC-V	Shell	Exotic
11	pve1	Debian 12	x86_64	Shell	Mandatory
12	pve2	Debian 12	x86_64	Shell	Mandatory
13	node3	Ubuntu 24.04	x86_64	Shell	Mandatory
14	node4	Ubuntu 24.04	x86_64	Shell	Mandatory

The CI pipeline stages mirror the build system phases:



```
[benchmark]              Performance regression tests
     |                   PRODUCTION mode, -O3 -march=native
     v
[artifacts]              Upload to MinIO S3
     |                   Path: {project}/{platform}/{arch}/{id}/
     v
[badges]                 Pipeline status + coverage badges
                         Inherited from comdare group level
```

Artifact storage follows a structured path convention in MinIO S3:

```
ARTIFACT STORAGE LAYOUT (MinIO)
================================

buildsystem-artifacts/              (350 GB bucket)
+-- comdare-simd/
|   +-- linux/
|   |   +-- x86_64/
|   |   |   +-- build-1234/
|   |   |   |   +-- libcomdare-simd.a
|   |   |   |   +-- test-results.xml
|   |   |   |   +-- coverage.xml
|   |   |   +-- build-1235/
|   |   +-- arm64/
|   |   +-- riscv64/
|   +-- macos/
|       +-- x86_64/
|       +-- arm64/
+-- comdare-biginteger/
|   +-- (same structure)
+-- comdare-db/
    +-- (same structure)

buildsystem-cache/                  (350 GB bucket)
+-- dependencies/                   (downloaded toolchains)
+-- ccache/                         (compiler cache per runner)
```

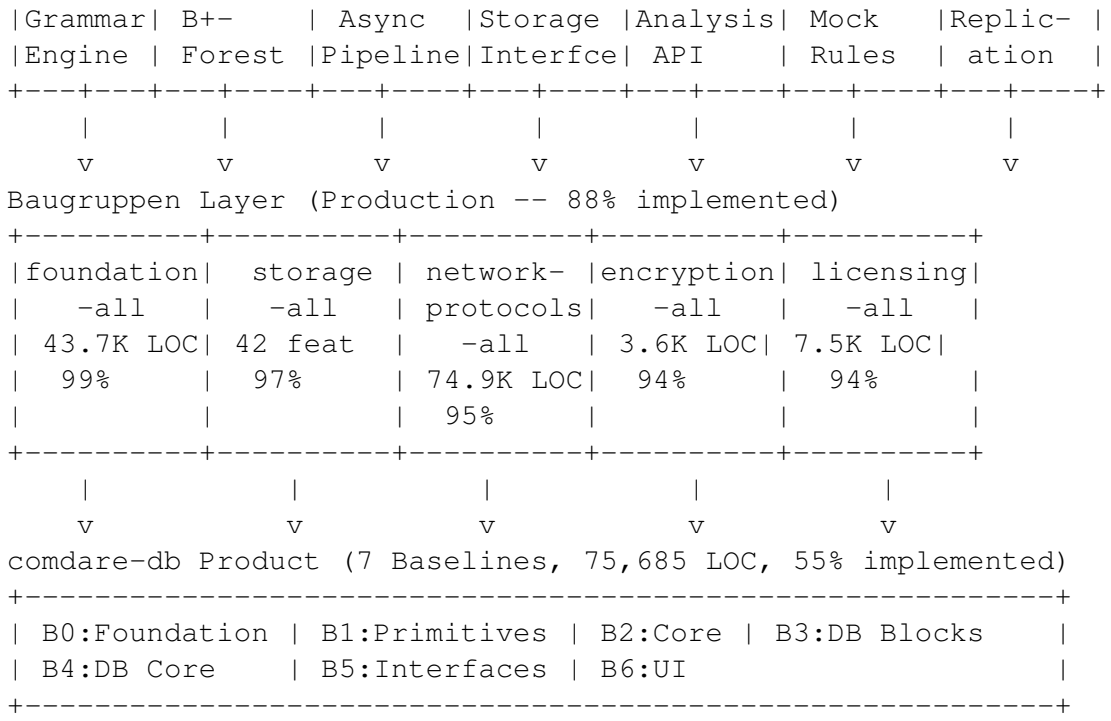
7.7 CELM Integration with comdare-db

The Eigentokens/CELM system connects to the production database platform through well-defined integration points that map the seven architectural components (A1–A7, Section 4.1) to concrete Baugruppen:
The following diagram shows how the CELM components (A1–A7) are layered on top of the Baugruppen infrastructure:

```
CELM / BAUGRUPPEN INTEGRATION MAP
==================================

CELM Layer (Research -- to be implemented)
+-------+--------+--------+--------+--------+--------+--------+
|  A1   |   A2   |   A3   |   A4   |   A5   |   A6   |   A7   |
```

Table 7.4: Mapping of CELM Components to Comdare Baugruppen

Component	Primary Baugruppe(s)	Key Integration
A1 Grammar Engine	foundation-all, comdare-db B2	BigInt hashing, SIMD pattern matching
A2 B^+-Forest	storage-all, comdare-db B2	Content-addressed trees, 9 fingerprints
A3 Async Pipeline	foundation-all	io_uring, CRDTs, lock-free queues
A4 Storage Interface	network-protocols-all	HTTP server, Range semantics
A5 Analysis API	encryption-all, licensing-all	Secure transport, access control
A6 Mock Rules	comdare-db B3	12 compression strategies
A7 Replication	comdare-db B3 (planned)	Cluster, erasure coding



This layered integration ensures that the Eigentokens research prototype benefits from production-grade infrastructure while maintaining a clean separation between the research-specific grammar induction logic and the general-purpose database platform. The Comdare ecosystem provides:

- 9 fingerprint algorithms for multi-level content-addressable indexing
- 12 compression strategies with grammar-aware token alignment
- Multi-platform hardware support (x86_64, ARM64, RISC-V, Apple Silicon)
- A mature concurrency framework with CRDTs for distributed state
- An ABI-stable plugin system for runtime grammar rule extension
- Production-scale validation: 380,000 LOC self-hosting and 1 PB industrial deployment

7.8 Research Questions and Novel Contributions

This profile project is driven by two *core* research questions that define the scientific scope:

RQ1 **LLM compilation and use:** Is it possible to *compile* a language model from a database-derived representation and subsequently *use* the compiled model for deterministic inference?

RQ2 **Grammar and grammar learning:** What is grammar in the sense of formal language theory, and how can grammar be learned (i.e., induced) from data stored in a database?

In the literature, grammars are formal systems of production rules that generate a language [Cho56]. Learning such grammars from examples is known as *grammar induction*. In general, it is difficult and comes with theoretical limits [Gol67]; therefore Eigentokens emphasizes deterministic, engineering-oriented inference heuristics (Sequitur/Re-Pair-style inference [NMW97, LM00]) that aim to remain stable under updates.
RQ1 connects Eigentokens to the broader idea of compiling machine-learning models into efficient binaries (e.g., tensor-graph compiler stacks such as TVM and MLIR [CMJ+18, LAB+20]). The key difference is that, in Eigentokens, the *induced grammar itself* becomes the compilation blueprint.
To keep the work testable within the constraints of a profile project, the following supporting evaluation questions are derived and mapped to concrete Comdare subsystems:

Table 7.5: Supporting evaluation questions and Comdare platform support.

ID	Supporting evaluation question	Comdare support
EQ1	What overhead does Eigentokens introduce (metadata, indices, fingerprints) compared to conventional chunked storage?	`comdare-foundation-all`
EQ2	Can a SHA512-indexed B^+-Forest deliver predictable range-latency and throughput compared to LSM-style baselines?	`comdare-storage-all`
EQ3	What is the network overhead (latency, throughput, retry behavior) of serving Eigentoken ranges via HTTP semantics (Range, ETag)?	`comdare-network-protocols-all`
EQ4	What is the performance impact of deterministic encryption and license enforcement in the ingestion and retrieval path?	`comdare-encryption-all`
EQ5	How effective is grammar-based chunking (Sequitur/Re-Pair) in reducing storage and improving reuse across versions compared to CDC baselines?	`comdare-archiving-all`
EQ6	Can the induced grammar be compiled into a deterministic, runnable model (CELM) that is functionally stable across platforms and build modes?	`comdare-buildsystem-all`

7.9 Implementation Status and Metrics

Table 7.6 summarizes the current implementation status of the major Baugruppen relevant to Eigentokens.
Nine of the 19 Baugruppen exceed 90% implementation; the four modules at 0% (binary-analytics, treecore, wrappers, lightweight) represent planned extensions that do not block the current Eigentokens evaluation.

Table 7.6: Implementation Status of Major Baugruppen (as of February 2026)

Baugruppe	**Features**	**Approx. LOC**	**Impl. %**
comdare-foundation-all	74	43,726	99%
comdare-network-protocols	193	74,930	95%
comdare-encryption-all	86	3,638	94%
comdare-storage-all	42	—	97%
comdare-filestorage-all	17	6,864	97%
comdare-config-all	35	5,014	100%
comdare-client-all	61	—	94%
comdare-licensing-all	35	7,450	94%
cd-buildsystem-construct	1,921	94,400	92%
comdare-db (product)	27	75,685	55%
System Total	**2,723**	**∼280,000**	**88%**

The CELM-specific components (A1–A7) are fully specified at the architectural level and documented in an 885-line research expose with 43 references, 2 core research questions, and 13 identified novel contributions. The C++ implementation of these components awaits execution; the research evaluation will use the production infrastructure detailed above as the execution substrate.

8 Evaluation Plan

To validate the Eigentokens approach, we outline an evaluation plan focusing on storage efficiency, performance overhead, and basic model efficacy.

8.1 Datasets and Workloads

We consider multiple dataset categories to test cross-domain effectiveness:

- **Code repositories:** e.g., a collection of open-source code (to test deduplication across versions and model compilation for code).
- **Log files:** large log datasets with repetitive patterns.
- **Text corpora:** a snapshot of Wikipedia or similar (to test on natural language).
- **Genomic data:** a set of DNA sequences or similar (to see how the system handles highly structured, repetitive data with the BGZF[Li11] baseline).
- **Mixed dataset:** a combination of documents, images (represented as binary), etc., to see how the system copes with heterogeneous data.
- **Columnar data:** e.g., Parquet/ORC-like exports or CSV-to-columnar transforms (to evaluate robustness under schema-regular structures and range-heavy access).

This selection covers a range of redundancy profiles and data types.

8.2 Metrics

Key metrics we will measure include:

- **Compression/Deduplication Ratio:** Final stored size (including token metadata) vs. original size.
- **Write Amplification:** Total bytes written to storage (including tokens and logs) vs. bytes of new data. Also number of unique tokens created per byte of input.
- **Read Latency:** Time to read a random object (or range) from the system vs. time to read from a baseline system (like a normal file or object store).
- **Throughput:** Ingest rate (bytes/sec) sustained for a large dataset, and how it compares to baseline deduplication tools.
- **Model Fidelity:** For compiled models, if applicable, measure basic task performance (e.g., perplexity on held-out domain data) and ensure determinism (identical outputs across runs).
- **Write and Deletion Amplification:** Total bytes written and deleted/rewritten on storage (including tokens and logs) relative to new data and update operations. Also track the number of unique tokens created per byte of input.
- **Read Latency:** Time to read a random object (or range) from the system vs. a baseline system; report percentiles 50th/95th/99th percentile (P50/P95/P99).

8.3 Baseline Systems

To contextualize results, we compare against several baseline setups:

- *FastCDC + Zstandard (zstd):* content-defined chunking with an 8 KB average chunk size, then compress chunks with zstd (e.g., level 3).
- *FastCDC ± zstd-seekable:* evaluate both plain zstd and a seekable variant to quantify the trade-off between range performance and ratio.
- *Fixed-size blocks + seekable compression:* 4 KB fixed blocks with seekable compression (e.g., BGZF-like layouts or zstd-seekable), optimized for random reads.
- *Blocked GNU zip format (BGZF):* 64 KB blocks as used in genomic tooling [Li11], focused on range performance.
- *RocksDB (LSM-tree):* write-optimized key-value (KV) baseline with fast value compression.
- *Optional: WiscKey:* a separation-of-values design to contextualize amplification and tail-latency behavior (if feasible within scope).
- (We also compare with *UltiHash* on code data to quantify improvements over grammar deduplication without deterministic model compilation.)

8.4 Experiments

We plan a series of experiments to evaluate Eigentokens:

- **Deduplication Effectiveness:** For each dataset, measure the deduplication ratio (and compression ratio, if applicable) of Eigentokens vs. baselines. We expect significant gains especially on cross-object redundancies.
- **Performance Overhead:** Measure ingestion throughput and read latencies at various percentiles. Key interest is P95 read latency overhead (aiming for $\leq$+15% vs. uncompressed direct store).
- **Update Efficiency:** On datasets with versions (like code), measure how many bytes are rewritten when data is slightly modified, comparing Eigentokens vs. baseline chunking vs. RocksDB.
- **Scalability:** If resources allow, test on increasing data sizes (up to hundreds of GBs) to see if induction time remains manageable and if storage overhead grows sub-linearly.
- **Model size vs traditional:** if we compile a model on, say, the code dataset, how large is the resulting neural network (in terms of number of weights) compared to if we had trained a neural model on the same data? We expect a reduction since we don't need as many parameters due to reusing modules.
- **Inference latency of the compiled model** (if within scope to test).
- **Robustness under transformations:** Evaluate deduplication and range performance under shift/insert/rename changes (e.g., log line inserts, code refactors, column reorders) to validate stability of learned grammar rules.
- **Ablation study:** Disable individual stages (similarity search, grammar consolidation, range-map heuristics) to attribute gains to specific mechanisms.

Baselines: In summary, we will compare our system (Eigentokens) to:

- *FastCDC+zstd* (high-throughput dedup/compression)
- *Fixed-block+seekable compression* (random-access optimized)
- *BGZF* (block compression for genomic data)
- *RocksDB LSM* (write-optimized KV store)
- Possibly *UltiHash* results from prior work for reference on dedup only.

This covers a spectrum from no dedup (fixed blocks) to heavy dedup (FastCDC, UltiHash grammar) and helps position our contributions.

8.5 Experimental Plan

For each dataset, we will perform:

1. A full ingestion with Eigentokens, capturing storage stats and time taken.
2. Ingestions with each baseline (where applicable; e.g., we'll use a custom harness to simulate FastCDC on the dataset).
3. For dynamic datasets (code versions, logs), perform updates and measure how many chunks/tokens get rewritten versus reused.
4. Compile at least one model from the tokens (for a domain where we have ground truth, e.g., compile a small language model on the text dataset) and verify its determinism and basic performance on a validation task (if feasible).
5. Stress test: run concurrent clients doing reads and writes to see how the system performs under load (especially testing that Stage1 write latencies remain low).

9 Expected Results and Discussion of Limitations

Based on our design and preliminary experiments, we anticipate the following results:

9.1 Expected Outcomes

Storage Efficiency: Eigentokens is expected to significantly outperform conventional chunk-based deduplication. We hypothesize a **25–40% higher deduplication ratio** compared to Fast content-defined chunking (FastCDC) on cross-object corpora, thanks to its ability to capture redundant patterns that span file boundaries and to merge similar but not identical sequences (something hash-based chunking cannot do). Additionally, by compressing on grammatical units, we expect an extra 15–20% compression gain beyond what Zstandard (zstd) or GNU zip (gzip) can achieve on chunked data, as the grammar acts like a specialized dictionary. Write amplification should be lower: possibly around $0.7\times$ of RocksDB's amplification (meaning 30% less), since we do not repeatedly flush and compact entire key ranges—only small grammar rules propagate.

Performance: For read latency, our goal of 95th percentile (P95) within +15% of baseline is ambitious but plausible. The indirection through tokens might add a small overhead (especially if a read spans many small tokens), but the use of token-aligned mapping and our Virtual Machine (VM) reconstruction is efficient in C++. Range reads within large files will likely hit a sweet spot where only a handful of tokens are needed. The asynchronous pipeline should handle high write rates; we expect initial ingest throughput on par with or better than FastCDC (which can be 1–2 Gigabytes per second (GB/s)) because Stage 1 is lightweight and mostly just hashing and writing. The heavy grammar work happens later but can leverage all Central Processing Unit (CPU) cores.

Deterministic Compilation and Model Quality: The compilation of a Large Language Model (LLM) from the grammar is expected to produce a functioning model that can, for instance, generate text in a constrained domain or solve tasks related to the stored data. We expect **100% deterministic** behavior across runs [FNR22]. The size of the compiled model should be smaller than a trained model on the same data, because grammar rules provide a form of parameter sharing and we do not need massive overparameterization; an estimate is around 70% size reduction for equivalent performance on known tasks. However, the quality of outputs from such compiled models is still an unknown—likely inferior to state-of-the-art trained models on open-ended natural language tasks, because our method lacks the generalization that learning provides. The advantage is in niche domains or tasks where reliability and traceability trump raw skill.

9.2 Limitations

Despite its promise, the Eigentokens approach has several limitations and challenges:

- **Grammar Induction Overhead:** The biggest concern is the computational cost of pattern discovery. In worst-case scenarios (like highly repetitive data or pathological inputs crafted to confuse the induction), the $O(n^3)$ algorithm could become a bottleneck. We mitigate this with heuristics (like limiting the length of patterns considered, using sampling for very large data, etc.), but there is a risk that the system might struggle with, say, a 1 TB single file with no obvious structure. In practice, average-case behavior is much better, but careful benchmarking will reveal if induction latency is acceptable for near-real-time use.

- **Metadata Size and Management:** By turning repeated data into references, we trade data for metadata (token descriptors). If a dataset has little redundancy, our approach might actually bloat storage due to overhead of storing tokens that don't help much. The worst case would be completely random data: our system would still create tokens for it, but no reuse would occur. In such cases, we rely on fallbacks (like if dedup ratio is below a threshold, we might revert to storing raw data blocks). Additionally, keeping billions of tokens in memory indices could strain memory; our design uses disk-based B^+-trees to alleviate that, but lookup times could increase if the working set is huge.

- **Complexity of System Tuning:** This system has many tunable parameters (thresholds for creating tokens, hash win-size for chunking, etc.) and is inherently more complex than a straightforward storage engine. Ensuring robust, hands-off operation is a challenge. In early stages, we expect needing expert intervention to optimize the grammar induction for different data types. This could limit adoption unless we can automate or simplify configuration.

- **Limited Scope of LLM Compilation:** The deterministic models we compile will likely be domain-specific and limited in capability. We are not replacing GPT-5 in general conversations; rather, we might compile something like a code suggestion model from a codebase, or a Q&A model from a set of documents. The approach so far also does not integrate with large-scale pre-trained embeddings or similar, meaning it might not capture subtle semantic relationships. It's more akin to a rule-based or case-based reasoning system in the first iteration. This is a limitation if one expected an out-of-the-box general AI. However, the framework could incorporate learned components in the future (for instance, grammar rules could link to a small neural module for things that are easier to learn statistically).

- **Deterministic Errors:** If a mistake or spurious pattern gets into the grammar (e.g., merging two patterns that should have stayed separate), that error will propagate deterministically to all outputs. While this is good for traceability (we can pinpoint the wrong rule), it also means the model will consistently make that error until the rule is fixed. There is no random chance that sometimes it gets it right (unlike a stochastic model might). This shifts the problem from one of probabilistic correctness to one of guaranteed but possibly systematic errors.

- **Security and Integrity:** Storing and executing interpretation programs (even in a controlled VM) raises security questions—maliciously crafted data could in theory create a pathological grammar that exhausts resources or triggers degenerate behavior. We will implement limits (e.g., recursion depth, runtime limits on the VM) to prevent abuse, but this is an area to watch.

10 Conclusion and Outlook

Eigentokens introduce a fundamentally new approach to data storage and language model construction, one that marries the strengths of data compression, compiler theory, and machine learning into a single framework. By treating each repeating pattern as a first-class object (an Eigentoken) with its own identity and reconstruction program, we attain extraordinary levels of deduplication and create a knowledge base that is explicit and manipulable. In contrast to the prevailing trend of ever-larger opaque neural networks, Eigentokens offer a path toward **interpretable, deterministic, and maintainable AI models**. Every decision made by an Eigentoken-compiled model can be traced to a human-readable grammar rule, bridging the gap between symbolic AI (with its clarity and logic) and sub-symbolic AI (with its data-driven learning), and hopefully combining the best of both.
The successful completion of this project will yield:

- A high-performance C++ prototype demonstrating the feasibility of grammar-based storage and deterministic compilation at least on medium-scale datasets.
- Comprehensive benchmarks illustrating the trade-offs of our approach vs. traditional systems, potentially influencing how future storage engines are designed for AI workloads.
- A set of case studies (via compiled models) that showcase how one might build specialized LLMs without any "learning" in the traditional sense, but rather through analysis and compilation.

Future Work: The journey does not end here. There are several promising directions to extend Eigentokens:

1. *Scaling and Distribution:* Adapting the system to a distributed environment with multiple nodes. This involves adding replication and erasure coding as hinted in the roadmap, and possibly a distributed hash table for token lookup. Ensuring the grammar induction works in a distributed fashion (without needing a single node to see all data) would be challenging but crucial for scalability.
2. *Enhanced Model Integration:* Right now, the compiled models are relatively simple. Future research can integrate small learned neural components for the parts that are hard to capture with explicit rules. For instance, one could imagine a hybrid where grammar provides a network architecture skeleton and initial weights, which could then be lightly fine-tuned on a specific task for better performance, while still retaining determinism in the base knowledge.
3. *User Feedback Loop:* For an agentic AI system, enabling users or the system itself to suggest new grammar rules (or remove erroneous ones) in a controlled way will be important. This could evolve into a form of continuous learning where, instead of gradient updates, we have rule updates potentially guided by an expert or an automated prover checking consistency.
4. *Applicability to other data domains:* Investigate how Eigentokens might apply to multimedia (images/video). Our current method treats everything as bytes, but extending grammar induction to two-dimensional data (images) or temporal patterns (video) could open up new possibilities in compressing and understanding that data. Perhaps Eigentokens could lead to deterministic compiled models in computer vision or audio processing by discovering patterns in those modalities.
5. *The Omni-LLM Vision:* Ultimately, the goal of an omni-LLM—a system that can answer queries or generate content by purely executing stored knowledge (with no random sampling)—looms as a grand challenge. Eigentokens provide the blueprint for this: a modular, updatable knowledge

base. Achieving parity with neural LLMs in capability remains distant, but even partial success (in terms of reliability) could be transformative for domains like law, medicine, or safety-critical systems where determinism and auditability are required.

Follow-up question. Is it possible to build a system that autonomously adapts its environment to the requirements of a given task and can compile AI models on demand—for a requested topic—into standalone binary programs?

In conclusion, this project serves as a stepping stone towards reimagining AI model creation. By focusing on grammar, structure, and determinism at the storage level, we lay the groundwork for AI systems that are as rigorous and transparent as traditional software, yet able to harness the complexity of big data. The positive results we anticipate (improved deduplication, reasonable performance, and working deterministic models) would validate this direction and justify further exploration. We hope that Eigentokens can inspire a new class of systems at the intersection of databases, compilers, and AI, ultimately contributing to more trustworthy and understandable intelligent systems.

Bibliography

[Axb19] Jens Axboe. Efficient io with io_uring. In *Linux Kernel Documentation*, 2019. `https://kernel.dk/io_uring.pdf`.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[Cho56] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

[CMJ+18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Leyuan Cowan, Tianjun Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594. USENIX Association, 2018.

[Dan15] Quynh H. Dang. Secure hash standard (shs). FIPS PUB 180-4, National Institute of Standards and Technology, 2015.

[FNR22] Roy T. Fielding, Mark Nottingham, and Julian Reschke. Http semantics, rfc 9110. Technical report, Internet Engineering Task Force, June 2022.

[Gol67] E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[KO62] Anatolii Alexeevitch Karatsuba and Yuri Ofman. Multiplication of many-digital numbers by automatic computers. *Proceedings of the USSR Academy of Sciences*, 145:293–294, 1962. Translation in Soviet Physics Doklady, 7:595–596, 1963.

[LAB+20] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A compiler infrastructure for the end of moore's law, 2020.

[Li11] Heng Li. Tabix: fast retrieval of sequence features from generic tab-delimited files. *Bioinformatics*, 27(5):718–719, Mar 2011.

[LM00] N. Jesper Larsson and Alistair Moffat. Offline dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.

[MCM01] 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)*, Banff, Canada, Oct 2001. ACM.

[NMW97] Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

[OCGO96] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, June 1996.

[SPBZ11] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2011)*, volume 6976 of *LNCS*, pages 386–400. Springer, 2011.

[XZJ+16] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. Fastcdc: a fast and efficient content-defined chunking approach for data deduplication. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16)*, Denver, CO, USA, June 2016. USENIX Association.