Architecture Decision Record

Git CMS

Database-Free Content Management via Git Plumbing

Author: James Ross
Version: 1.0.0
Date: 2026-01-11

Prepared for Engineering Review

Contents

1 Introduction & Goals

1.1 Project Overview

git-cms is a serverless, database-free Content Management System that treats Git's object store as a distributed, cryptographically verifiable document database. Instead of storing content in traditional databases (SQL or NoSQL), it leverages Git's Merkle DAG to create an append-only ledger for articles, metadata, and encrypted assets.

The fundamental innovation: `git push` becomes the API endpoint.

1.2 Fundamental Requirements

1.2.1 FR-1: Zero-Database Architecture

The system MUST NOT depend on external database systems (SQL, NoSQL, or key-value stores). All persistent state resides within Git's native object store (`.git/objects`).

Rationale: Eliminates operational complexity, deployment dependencies, and schema migration challenges inherent to traditional database-backed CMSs.

1.2.2 FR-2: Cryptographic Verifiability

Every content mutation MUST be recorded as a Git commit with cryptographic integrity guarantees via SHA-1 hashing (with optional GPG signing for non-repudiation).

Rationale: Provides immutable audit trails and tamper detection without additional infrastructure.

1.2.3 FR-3: Fast-Forward Only Publishing

The publish operation MUST enforce strict linear history (fast-forward only) to prevent rewriting published content.

Rationale: Guarantees provenance and prevents content manipulation after publication.

1.2.4 FR-4: Client-Side Encryption

All uploaded assets MUST be encrypted client-side (AES-256-GCM) before touching the repository.

Rationale: Achieves row-level security without database-level access controls. The Git gateway receives only opaque encrypted blobs.

1.2.5 FR-5: Infinite Point-in-Time Recovery

Users MUST be able to access any historical version of any article without data loss.

Rationale: Git's DAG structure provides this naturally; the CMS simply exposes it as a first-class feature.

1.3 Quality Goals

Prio	Attribute	Description	Measurement
1	Security	Cryptographic integrity, signed commits	GPG verification, AES-256 strength
2	Simplicity	Minimal dependencies, composable architecture	Lines of code, dependency count
3	Auditability	Complete provenance of all content changes	Git log completeness
4	Performance	Sub-second reads for blog workloads	Response time for `readArticle()`
5	Portability	Multi-runtime support (Node, Bun, Deno)	Test suite pass rate

Table 1: Quality goals and their measurements.

1.4 Non-Goals

This system is **intentionally NOT designed for**:

- **High-velocity writes:** Content publishing happens in minutes/hours, not milliseconds.
- **Complex queries:** No SQL-like JOINs or aggregations. Queries are limited to ref enumeration and commit message parsing.
- **Large-scale collaboration:** Designed for single-author or small-team blogs.
- **Real-time updates:** Publishing is atomic but not instantaneous.

2 Constraints

2.1 Technical Constraints

TC-1: Git's Content Addressability Model
Git uses SHA-1 hashing for object addressing. While SHA-1 has known collision vulnerabilities, Git is transitioning to SHA-256. The system assumes SHA-1 is "good enough" for content addressing (not for security-critical signing).

Mitigation: Use GPG signing (`CMS_SIGN=1`) for cryptographic non-repudiation.

TC-2: Filesystem I/O Performance
All Git operations are ultimately filesystem operations. Performance is bounded by disk I/O, especially for large repositories.

Mitigation: Content is stored as commit messages (small), not files (large). Asset chunking (256KB) reduces blob size.

TC-3: POSIX Shell Dependency
The `@git-stunts/plumbing` module executes Git via shell commands (`child_process.spawn`). This requires a POSIX-compliant shell and Git CLI.

Mitigation: All tests run in Docker (Alpine Linux) to ensure consistent environments.

TC-4: No Database Indexes
Traditional databases provide B-tree indexes for fast lookups. Git's ref enumeration is linear (`O(n)` for listing all refs in a namespace).

Mitigation: Use ref namespaces strategically (e.g., `refs/_blog/articles/<slug>`) to avoid polluting the global ref space.

2.2 Regulatory Constraints

RC-1: GDPR Right to Erasure
Git's immutability conflicts with GDPR's "right to be forgotten." Deleting a commit requires rewriting history, which breaks cryptographic integrity.

Mitigation: Use encrypted assets with key rotation. Deleting the encryption key renders historical content unreadable without altering Git history.

RC-2: Cryptographic Export Restrictions
AES-256-GCM encryption may face export restrictions in certain jurisdictions.

Mitigation: The `@git-stunts/vault` module uses Node's built-in `crypto` module, which is widely available.

2.3 Operational Constraints

OC-1: Single-Writer Assumption
Git's ref updates are atomic *locally* but not across distributed clones. Concurrent writes to the same ref can cause conflicts.

Mitigation: Use **git-stargate** (a companion project) to enforce serialized writes via SSH.

OC-2: Repository Growth
Every draft save creates a new commit. Repositories can grow unbounded over time.

Mitigation: Use `git gc` aggressively. Consider ref pruning for old drafts.

3 Context & Scope

3.1 System Context Diagram

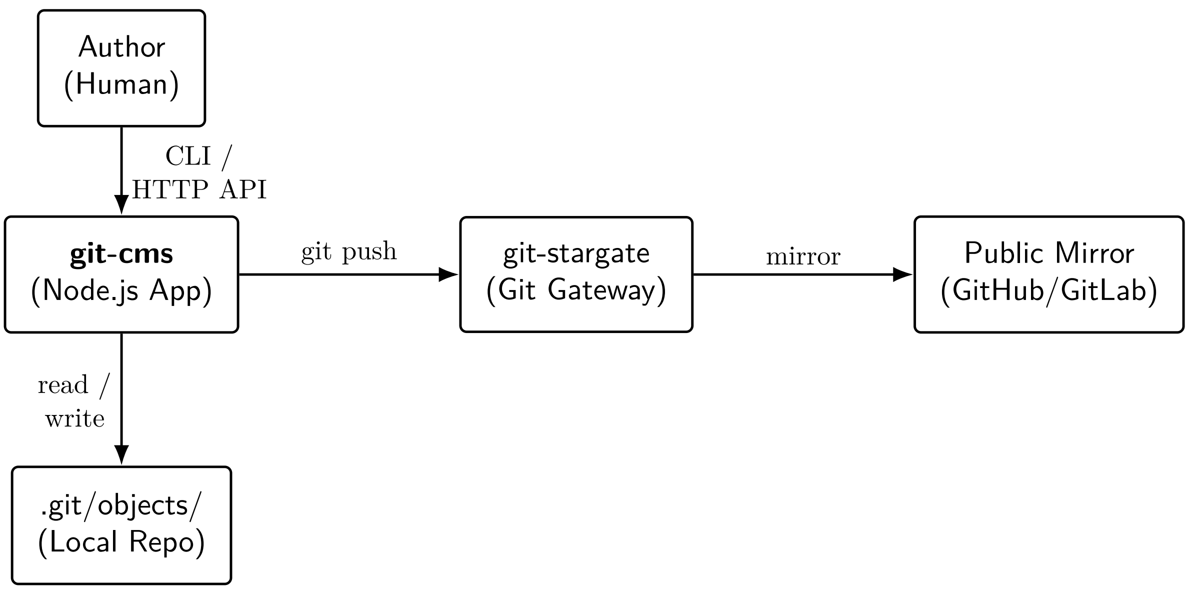

Figure 1: System context diagram showing the high-level relationship between the Author, Git CMS, and external components.

3.2 External Interfaces

3.2.1 Interface 1: CLI (Binary)

- **Entry Point:** `bin/git-cms.js`
- **Commands:** `draft`, `publish`, `list`, `show`, `serve`
- **Protocol:** POSIX command-line arguments
- **Example:**

```
echo "# Hello World" | git cms draft hello-world "My First Post"
```

3.2.2 Interface 2: HTTP API (REST)

- **Server:** `src/server/index.js`
- **Port:** 4638 (configurable via `PORT` env var)
- **Endpoints:**
  - `POST /api/cms/snapshot` – Save draft
  - `POST /api/cms/publish` – Publish article
  - `GET /api/cms/list` – List articles
  - `GET /api/cms/show?slug=<slug>` – Read article
- **Authentication:** None (assumes private network or SSH tunneling)

3.2.3 Interface 3: Git Plumbing (Shell)

- **Protocol:** Git CLI commands via `child_process.spawn`
- **Critical Commands:**
  - `git commit-tree` – Create commits on empty trees
  - `git update-ref` – Atomic ref updates
  - `git for-each-ref` – List refs in namespace
  - `git cat-file` – Read commit messages

3.2.4 Interface 4: OS Keychain (Secrets)

- **Platforms:**
  - macOS: `security` tool
  - Linux: `secret-tool` (GNOME Keyring)
  - Windows: `CredentialManager` (PowerShell)
- **Purpose:** Store AES-256-GCM encryption keys for assets

3.3 Scope Boundaries

3.3.1 In Scope

- Article drafting, editing, and publishing
- Encrypted asset storage (images, PDFs)
- Full version history via Git log
- CLI and HTTP API access
- Multi-runtime support (Node, Bun, Deno)

3.3.2 Out of Scope

- **User Authentication:** Delegated to git-stargate or SSH
- **Search Indexing:** No full-text search (external indexer required)
- **Media Transcoding:** Assets stored as-is

- **Real-Time Collaboration:** No OT or CRDTs
- **Analytics:** No built-in tracking

4 Solution Strategy

4.1 Core Architectural Principles

P-1: Composition over Inheritance The system is built from **five independent L̈ego Blockm̈odules** (`@git-stunts/*`), each with a single responsibility. These modules are composed in `CmsService` to create higher-order functionality.

Benefit: Each module can be tested, versioned, and published independently.

P-2: Hexagonal Architecture (Ports & Adapters) The domain layer (`CmsService`) depends on abstractions (`GitPlumbing`, `TrailerCodec`), not implementations. This allows swapping out Git for other backends (e.g., a pure JavaScript implementation for testing).

Benefit: Decouples domain logic from infrastructure concerns.

P-3: Content Addressability Assets are stored by their SHA-1 hash, enabling automatic deduplication. If two articles reference the same image, it's stored once.

Benefit: Reduces repository bloat.

P-4: Cryptographic Integrity Every operation produces a cryptographically signed commit (when `CMS_SIGN=1`). The Merkle DAG ensures tamper detection.

Benefit: Audit trails are mathematically verifiable, not just trust-based.

4.2 Solution Approach: The Ëmpty TreeS̈tunt

The Problem Traditional CMSs store content in database rows. Git is designed to track *files*, not arbitrary data. Storing blog posts as files (e.g., `posts/hello-world.md`) clutters the working directory and causes merge conflicts.

The Solution Store content as **commit messages on empty trees**, not as files. Every article is a commit that points to the well-known empty tree (`4b825dc642cb6eb9a060e54bf8d69288fbee4904`).

How It Works:

1. Encode the article (title, body, metadata) into a Git commit message using RFC 822 trailers.
2. Create a commit that points to the empty tree (no files touched).
3. Update a ref (e.g., `refs/_blog/articles/hello-world`) to point to this commit.

Result: The repository's working directory remains clean. All content lives in `.git/objects/` and `.git/refs/`.

Architectural Pattern: Event Sourcing Each draft save creates a new commit. The c̈urrentärticle is the ref's tip, but the full history is a linked list of commits.

Benefit: Point-in-time recovery is trivial (`git log refs/_blog/articles/<slug>`).

4.3 Key Design Decisions

D-1: Why Commit Messages, Not Blobs? Alternative: Store articles as Git blobs and reference them via trees.
Decision: Use commit messages.
Rationale: Commits have parent pointers (version history) and support GPG signing (non-repudiation). Blobs are opaque; messages are human-readable.

D-2: Why Trailers, Not JSON? Alternative: Store `{"title": "Hello", ...}` as the message.
Decision: Use RFC 822 trailers.
Rationale: Trailers are Git-native, human-readable, and diff-friendly. Backward parsing is efficient.

D-3: Why Encrypt Assets, Not Repos? Alternative: Use `git-crypt` for the whole repo.
Decision: Encrypt individual assets client-side.
Rationale: Granular control; the gateway never sees plaintext.

5 Building Block View

5.1 Level 1: System Decomposition

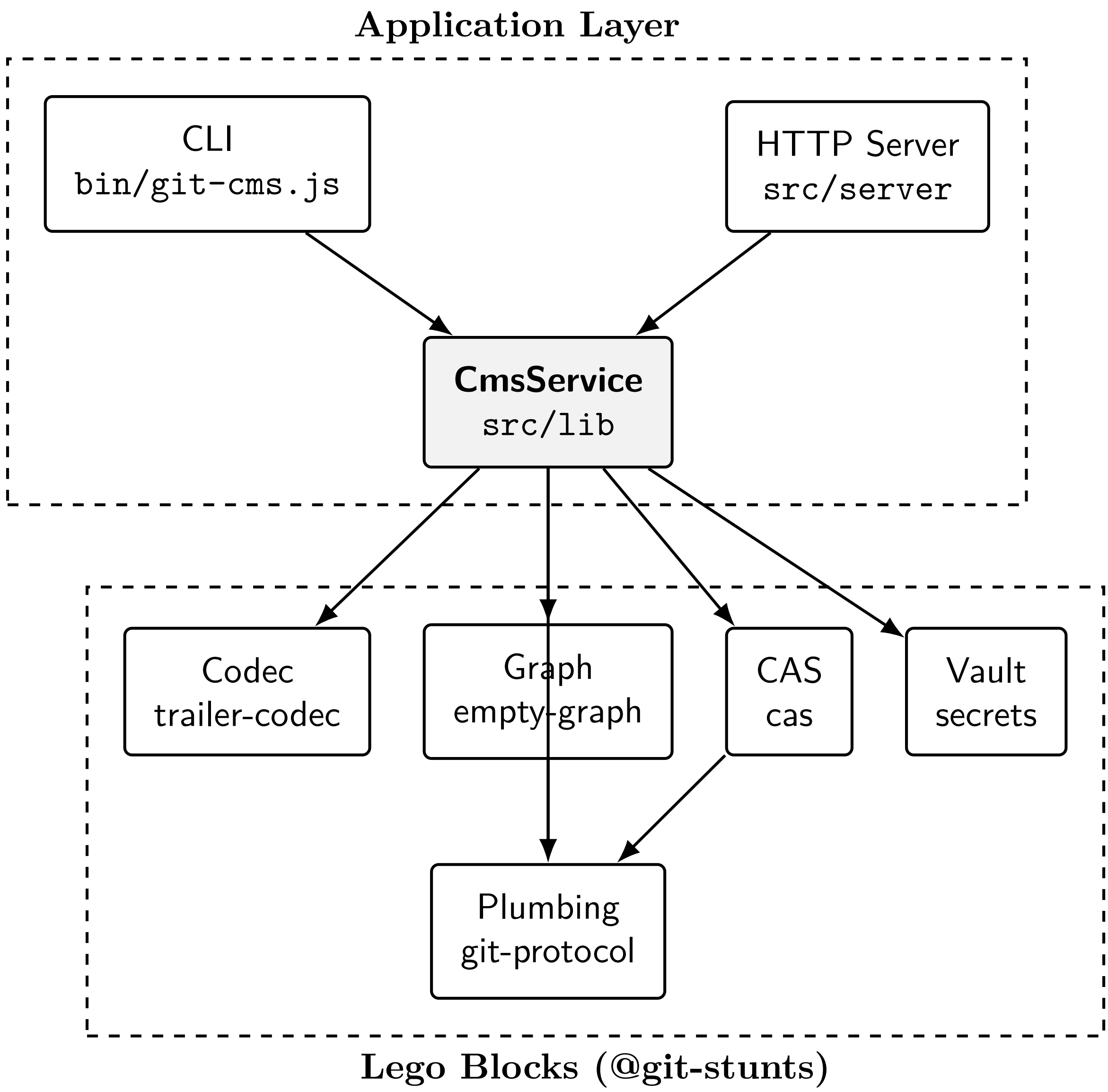

Figure 2: System decomposition showing the interaction between the application layer and the independent Lego block modules.

5.2 Level 2: Lego Block Responsibilities

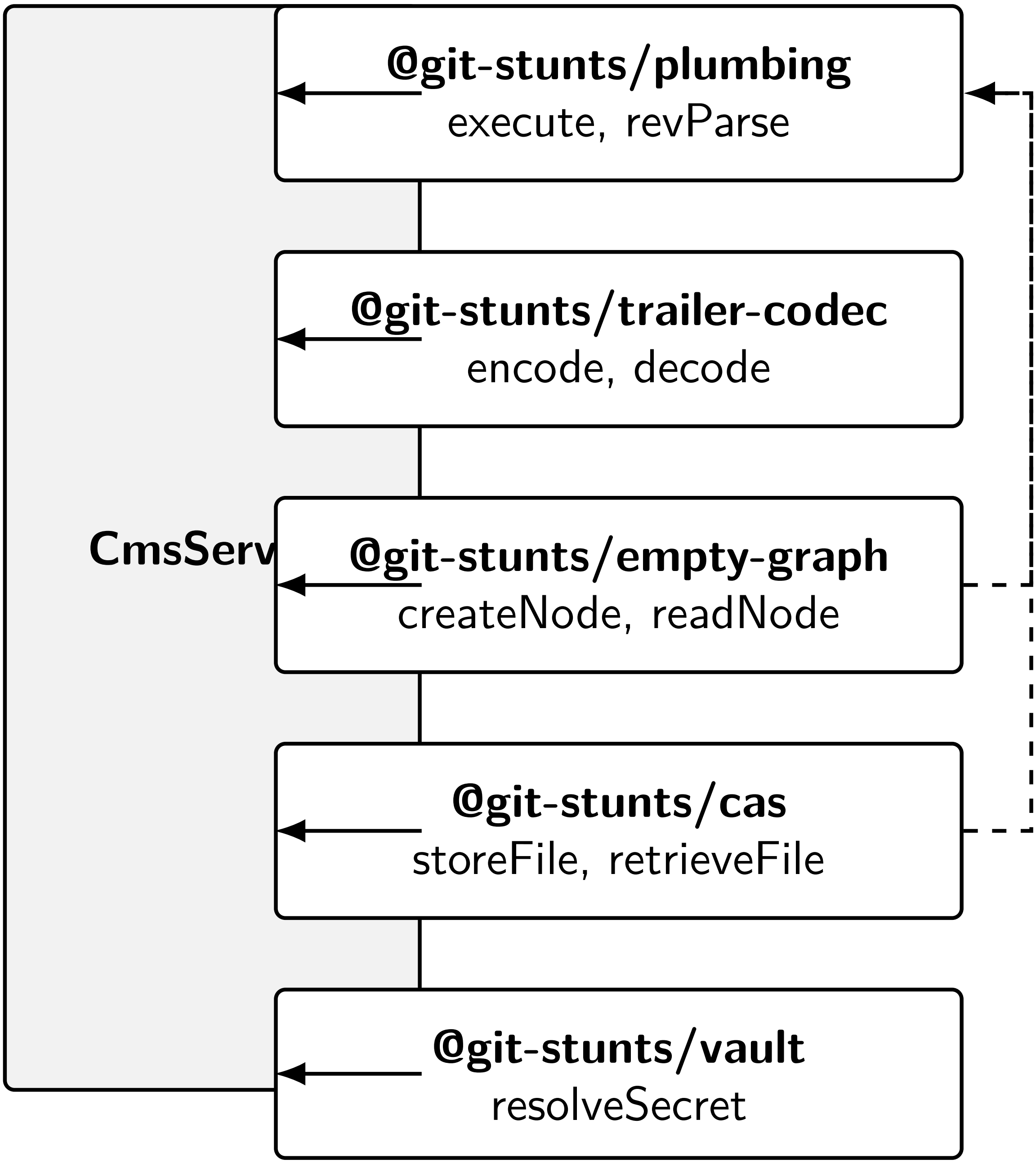

Figure 3: Detailed responsibilities and API surfaces of the Git CMS modules.

6 Runtime View

6.1 Scenario 1: Create Draft Article

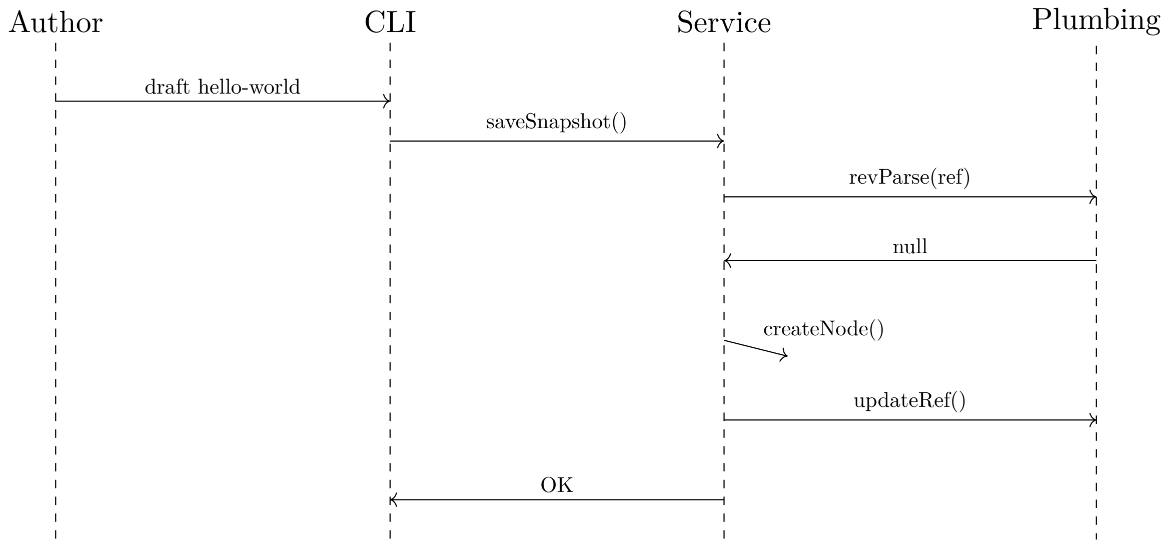

Figure 4: Sequence diagram for creating a new draft article.

6.2 Scenario 2: Publish Article

Publishing is **just a ref copy**. No new commits are created. This operation is idempotent and enforces fast-forward updates.

6.3 Scenario 3: Upload Encrypted Asset

The system splits files into 256KB chunks, encrypts them via AES-256-GCM, and stores them as Git blobs. The plaintext never touches the object store.

6.4 Scenario 4: List All Published Articles

Listing articles involves a linear scan of the ref namespace (`O(n)`). For large workloads, an external index is recommended.

7 Deployment View

7.1 Topology 1: Single-Author Local Blog

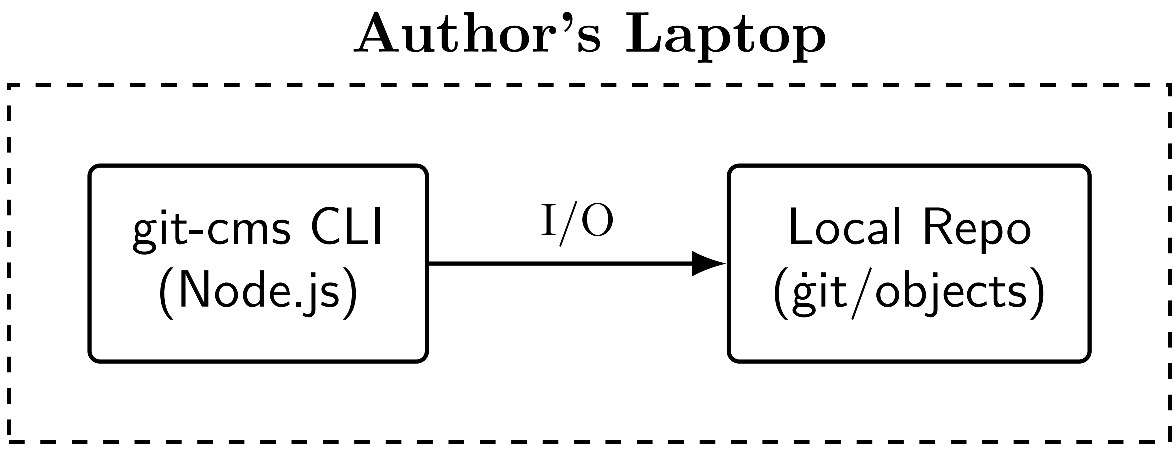

Figure 5: Local deployment topology.

7.2 Topology 2: Team Blog with Stargate Gateway

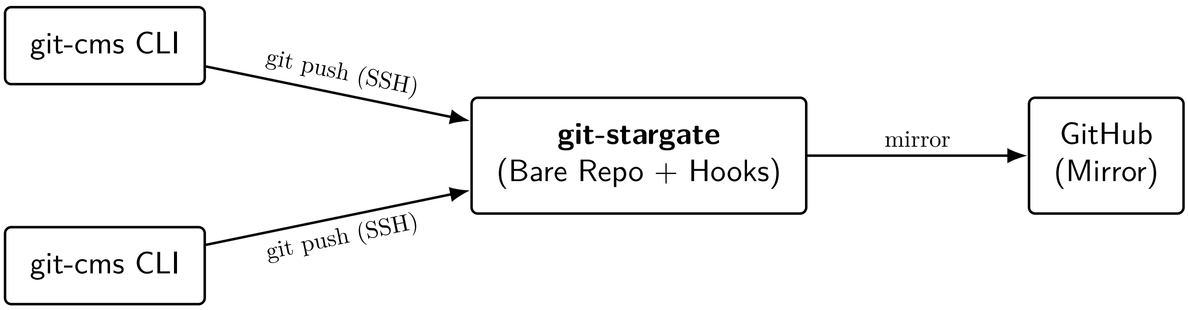

Figure 6: Collaborative deployment topology using a central gateway.

7.3 Topology 3: Dockerized Development

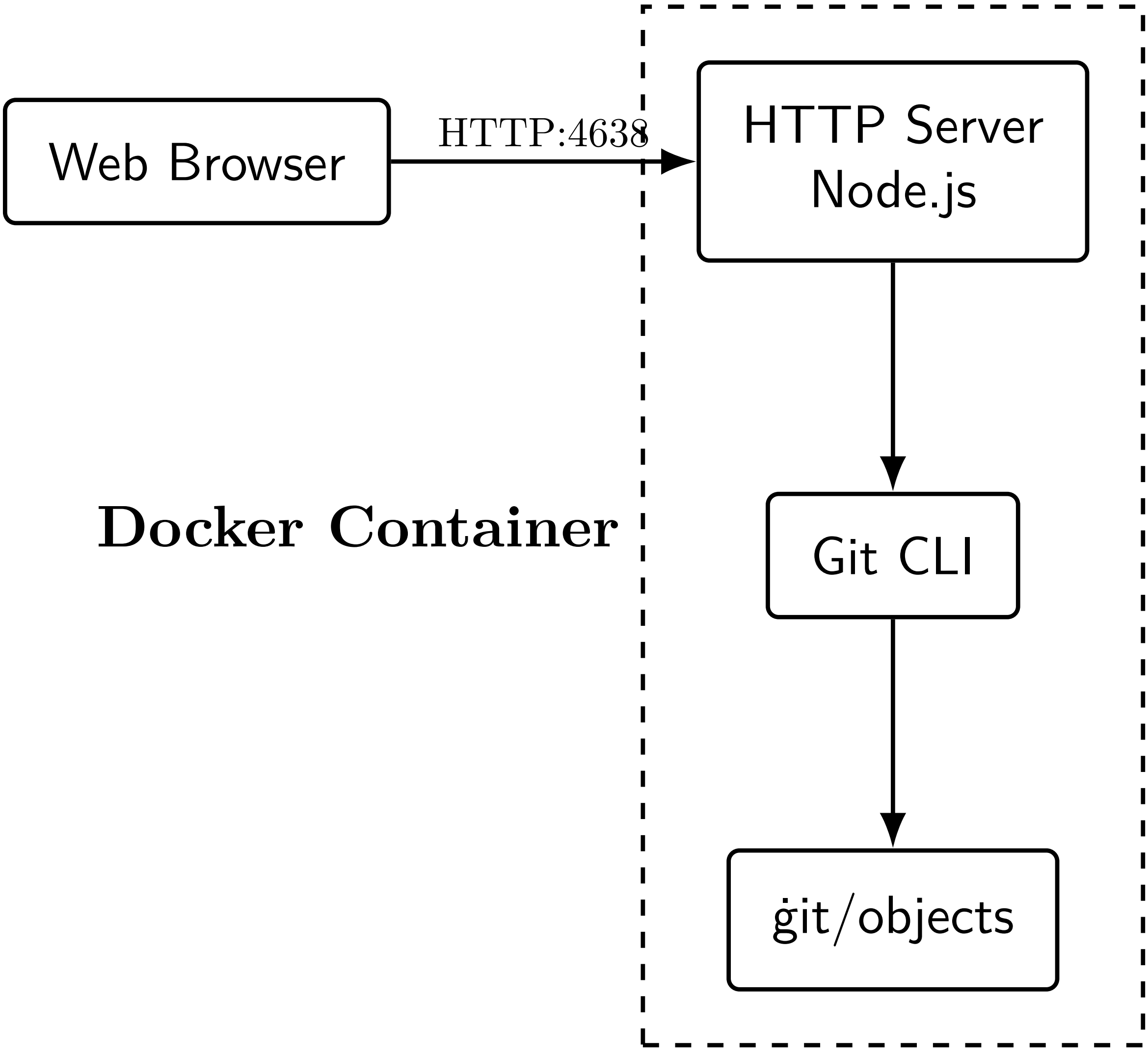

Figure 7: Dockerized development topology.

8 Crosscutting Concepts

8.1 Concept 1: Merkle DAG as Event Log

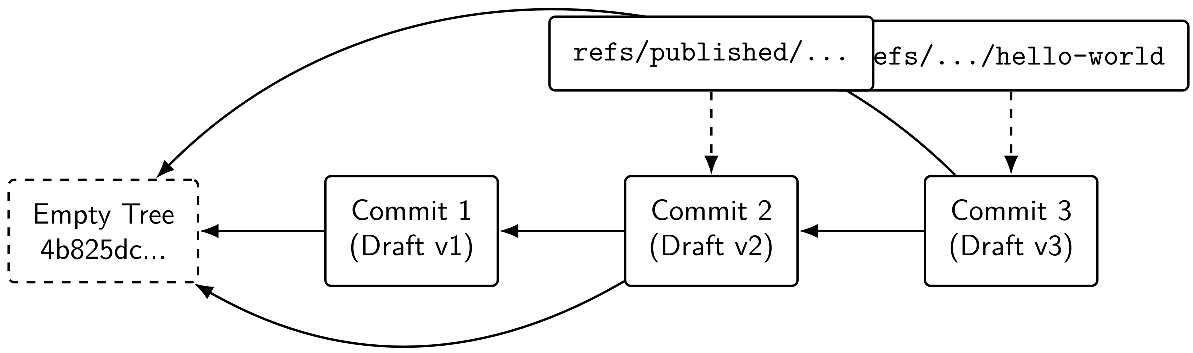

Figure 8: The Merkle DAG structure acting as an immutable event log.

8.2 Concept 2: Compare-and-Swap (CAS)

The system uses `git update-ref <ref> <newSHA> <oldSHA>` to ensure atomic updates and prevent race conditions. If the `oldSHA` has changed since it was last read, the update is rejected.

8.3 Concept 3: Client-Side Encryption

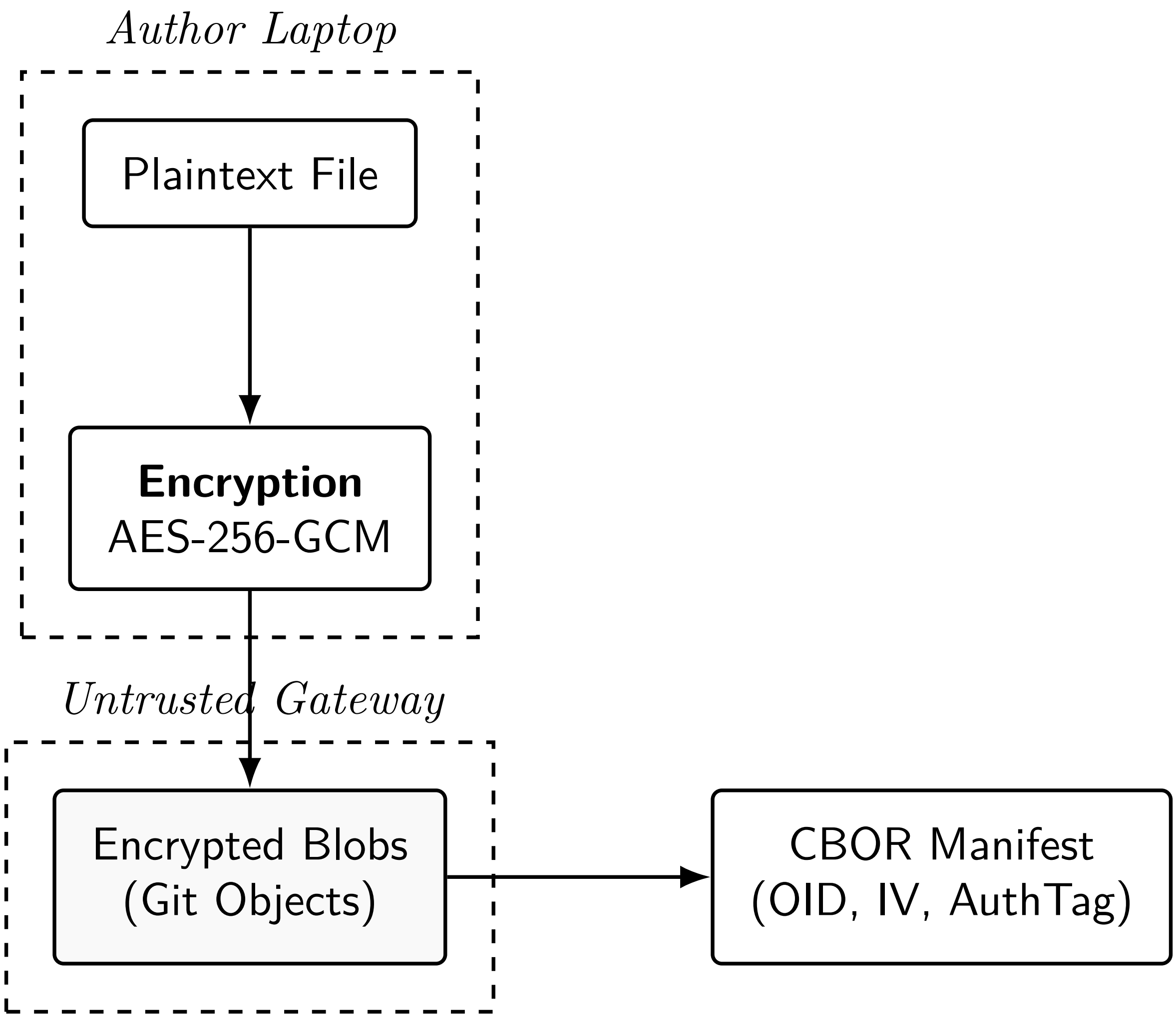

Figure 9: End-to-end encryption pipeline for assets.

9 Architectural Decisions

ADR-001: Use Commit Messages, Not Files

Context: Need to store articles in Git without polluting the working directory or causing merge conflicts on files.

Decision: Store article content (title, body, trailers) as Git commit messages pointing to the canonical empty tree (`4b825dc...`).

Rationale:

- Commits have parent pointers, enabling native version history.
- Commits support GPG signing for non-repudiation.
- Keeps the working directory completely clean for application code.

Status: Accepted.

ADR-002: Use RFC 822 Trailers, Not JSON

Context: Need structured metadata (Status, Author, etc.) inside commit messages.

Decision: Use RFC 822 trailers (key-value pairs at the end of the message).

Rationale:

- Git-native format (compatible with `git interpret-trailers`).
- Human-readable and extremely diff-friendly.
- Faster to parse from the end of the message.

Status: Accepted.

ADR-003: Fast-Forward Only Publishing

Context: Prevent published content from being altered or rewritten after release.

Decision: The publishing operation must be a strict fast-forward from the draft ref to the published ref.

Rationale: Guarantees that the exact same commit SHA that was reviewed/drafted is the one being published.

Status: Accepted.

ADR-004: Client-Side Encryption for Assets

Context: Git gateways or mirror repositories may be untrusted.

Decision: Encrypt all binary assets (images, PDFs) client-side using AES-256-GCM before uploading.

Rationale: defense-in-depth; the gateway only ever receives opaque encrypted blobs and an authenticated manifest.

Status: Accepted.

10 Quality Requirements

10.1 Quality Tree

The primary quality attributes for Git CMS are prioritized as follows:

1. **Security:** Cryptographic integrity and asset confidentiality.

2. **Simplicity:** Zero external database dependencies.

3. **Auditability:** Full provenance via Git's Merkle DAG.

4. **Performance:** Sub-second reads for standard blog workloads.

10.2 Quality Scenarios

10.2.1 QS-1: Tamper Detection

Scenario: An attacker modifies a published article directly on the Git gateway.
Stimulus: Malicious rewrite of Git history (`filter-branch`).

Response: The Merkle DAG checksum mismatch is immediately detected by any client pulling the update.
Metric: 100% detection of unauthorized history rewrites.

10.2.2 QS-2: Confidentiality

Scenario: A repository mirror is compromised.
Stimulus: Attacker attempts to view private image assets.
Response: Only AES-256-GCM ciphertext is visible; plaintext remains unrecoverable without the client-side key.
Metric: 0% leakage of plaintext assets.

11 Risks & Technical Debt

11.1 Risk 1: SHA-1 Collision

Git's reliance on SHA-1 is a known cryptographic risk. While the likelihood of a practical attack on a blog is low, the system should monitor Git's transition to SHA-256.

11.2 Risk 2: Repository Growth

Every draft save creates a permanent commit. Over years of active use, the object store could grow significantly. Regular `git gc -aggressive` and ref pruning strategies are needed.

11.3 Technical Debt Summary

Item	Priority	Impact
Automated ref pruning	High	Reduces repo bloat
Retry logic for CAS conflicts	Medium	Improves concurrent editing
External index for large ref counts	Medium	Improves `listArticles` performance

12 Glossary

AES-256-GCM Advanced Encryption Standard with 256-bit keys in Galois/Counter Mode.

Bare Repository A Git repository without a working directory, typically used on servers.

CAS Content-Addressable Store (or Compare-and-Swap, depending on context).

Commit A snapshot of the repository at a point in time.

Empty Tree The unique OID (`4b825dc...`) of a tree containing zero files.

Merkle DAG A directed acyclic graph where each node is identified by the hash of its content.

Ref A pointer to a Git object (e.g., branch, tag, or article slug).

Trailer RFC 822 metadata at the end of a commit message.

A Appendix A: Example Commands

A.1 Draft an Article

```
echo "# My First Post" | git cms draft hello-world "My First Post"
```

A.2 Publish an Article

```
git cms publish hello-world
```

A.3 List All Drafts

```
git cms list
```

A.4 Upload Asset

```
git cms upload hello-world image.png
```

B Appendix B: Directory Structure

```
git-cms/
+-- bin/
|   +-- git-cms.js                # CLI entry point
+-- src/
|   +-- lib/
|   |   +-- CmsService.js         # Core orchestrator
|   +-- server/
|       +-- index.js              # HTTP API server
+-- test/
|   +-- git.test.js               # Integration tests
|   +-- e2e/                      # Playwright tests
+-- public/                       # Static admin UI
+-- docs/                         # Documentation
```

C Appendix C: Related Projects

- **git-stargate:** Git gateway for enforcingFF-only and signing.
- **git-stunts:** Lego blocks for Git plumbing.

D Appendix D: References

1. Git Internals (Pro Git Book)
2. RFC 822 (Internet Message Format)
3. AES-GCM (NIST SP 800-38D)

E Appendix D: References

1. **Git Internals (Pro Git Book):** `https://git-scm.com/book/en/v2/Git-Internals-Plumbing-and-Porcelain`

2. **RFC 822 (Internet Message Format):** `https://tools.ietf.org/html/rfc822`

3. **AES-GCM (NIST SP 800-38D):** `https://csrc.nist.gov/publications/detail/sp/800-38d/final`