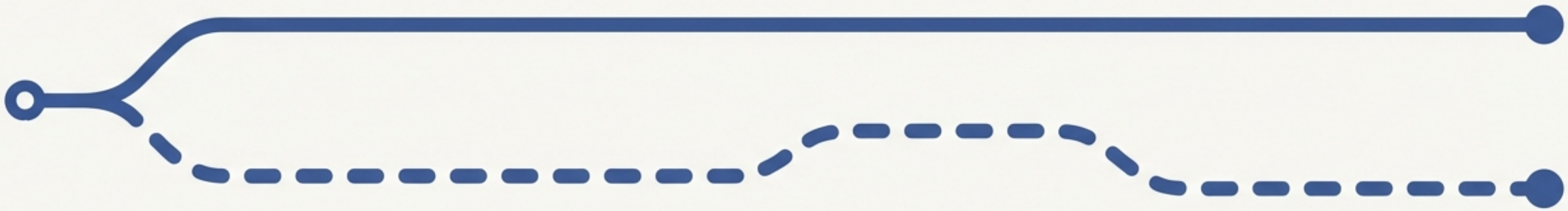
The Dual Save Pattern

Never Lose History. Never Lose Speed.



An elegant, lightweight pattern for efficient data versioning in modern applications.

The Engineer's Dilemma: Juggling Speed and History

How do we get instant access to the *current* state of our data, while also preserving a *complete* history of every change?

Fast Access to Current State

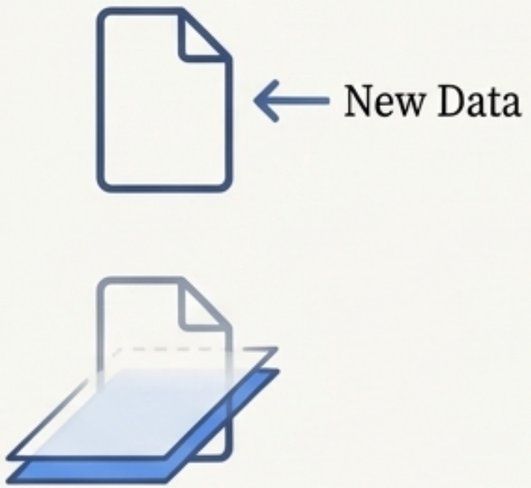
Applications require a quick way to find the most recent version of a record, representing the current reality.

Complete Historical Record

It's valuable to retain past versions to see how data changed over time—for debugging, audits, or understanding trends.

Why Naive Approaches Fall Short

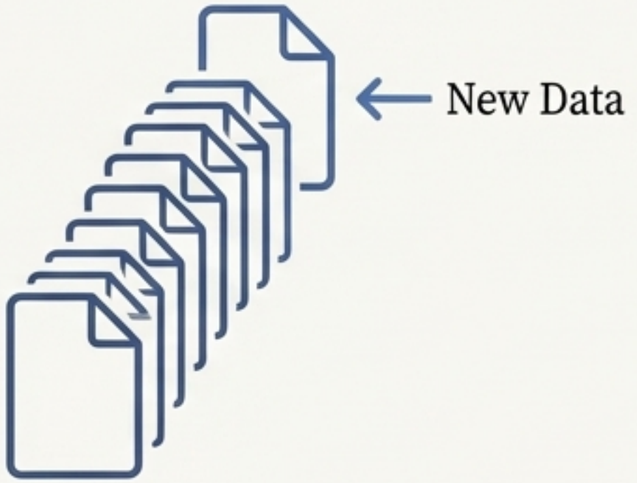
Overwrite-in-Place



How it Works: The latest version of data simply replaces the old one.

- **Pro:** Blazing fast reads for the current state.
- **Con:** History is permanently lost. No audit trail, no provenance.

Append-only Archive

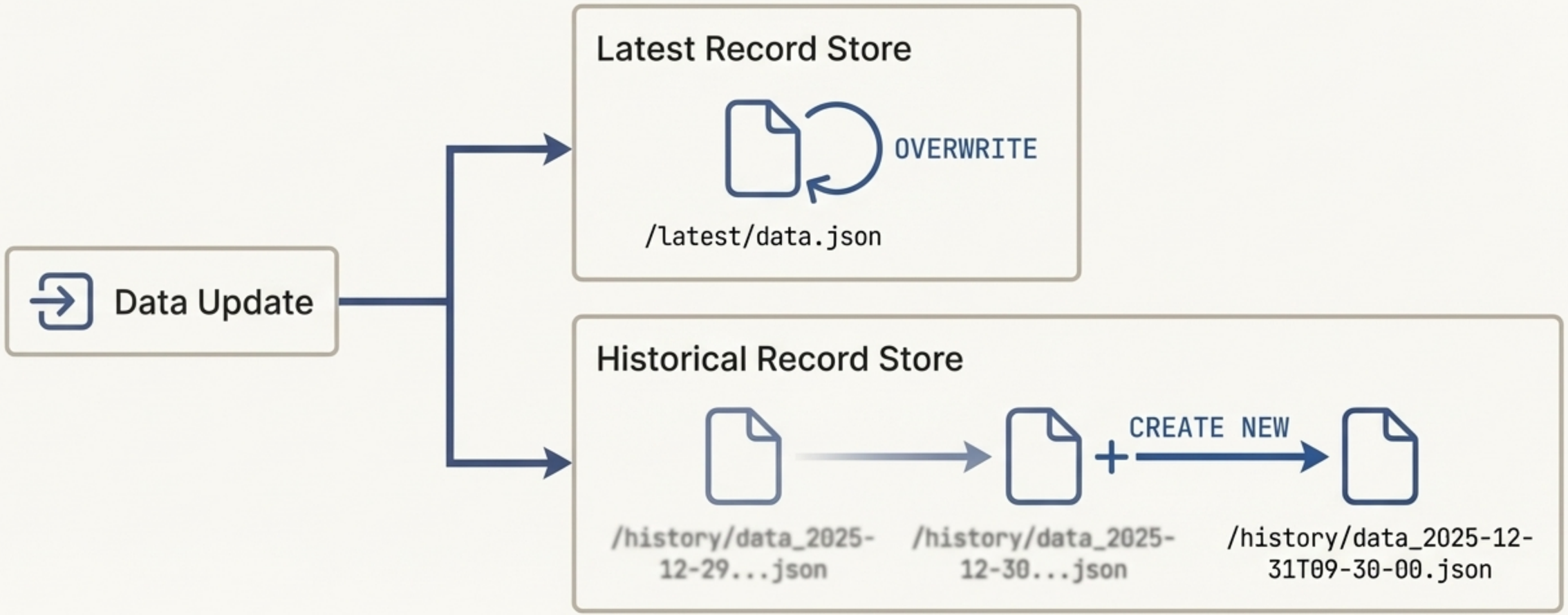


How it Works: Every single version is appended to a growing log or archive.

- **Pro:** A complete and immutable history is preserved.
- **Con:** Finding the “latest” version becomes slow and complex, requiring scans or sorting.

The Solution: The Dual Save Pattern

On every update, save the data twice: once as the latest version, and once as a permanent, timestamped record.



The Core Mechanic: Two distinct roles, one single truth.

The Latest Record

A single, mutable pointer to the truth *now*. Stored in a known, deterministic location for instant retrieval. Any process needing the current state knows exactly to look.

The Historical Records

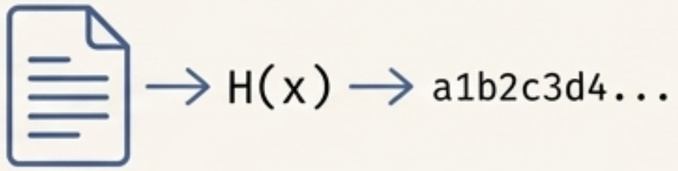
An immutable, chronological log of the past. Each significant change is preserved as a unique, timestamped snapshot, creating a complete audit trail.

A single, mutable pointer to the truth *now*. Stored in a known, deterministic location for instant retrieval. Any process needing the current state knows exactly where to look.

The Step-by-Step Process, Part 1: The Efficiency Check

1. Compute Content Hash

An incoming data update is received. First, compute a hash (e.g., SHA-256) of its meaningful content. This creates a unique signature.



2. Compare with Latest

Retrieve the hash of the current 'latest' version and compare it with the new hash.

3. Decision Point

Has the meaningful content actually changed?

Hashes Match

Stop the process.
No new version is created. This avoids duplicating unchanged data.

Hashes Differ

Proceed to write.

The Step-by-Step Process, Part 2: The Dual Write

Hashes Differ

4. Save to Latest

Overwrite the record in the “latest” location with the new data. Any subsequent read will now get this updated state.

Example: Write to `/latest/data.json` or `UPDATE` the row in the `current_records` table.

5. Save to History

Save a copy of the new data as a new, permanent record identified by a timestamp or version ID.

Example: Create `/history/data_2025-12-31T09-30-00.json` or `INSERT` a new row into the `history_records` table.

End

The Payoff: Performance and Provenance

Instant Access to Current Data

The ‘latest’ record is stored in a predictable location. This allows any process to fetch the current state instantly without scanning archives or computing differences, improving performance and simplifying client logic.

Complete Historical Provenance

All distinct states of the data are preserved. This provides full traceability of changes, enabling you to answer questions about past states, debug issues, or perform trend analysis. The data's lineage is recorded step-by-step.

The Payoff: Efficiency and Simplicity

Built-in Storage Efficiency

By leveraging content hashing, a new version is only stored when data *actually* changes. This avoids cluttering history with duplicate entries and significantly reduces long-term storage costs compared to naive versioning.

Elegant and Simple Design

The pattern is conceptually simple and requires no complex frameworks. It can be implemented with basic file operations or database writes, making it easy to adopt and reason about in many different technology stacks.

Use Case: Infrastructure and Configuration Tracking

Scenario: Monitoring AWS EC2 Security Group rules.

Latest View

For `prod-web-01`, ports 80 and 443 ✅ are open. ✅

This is used for real-time security checks.

Historical View

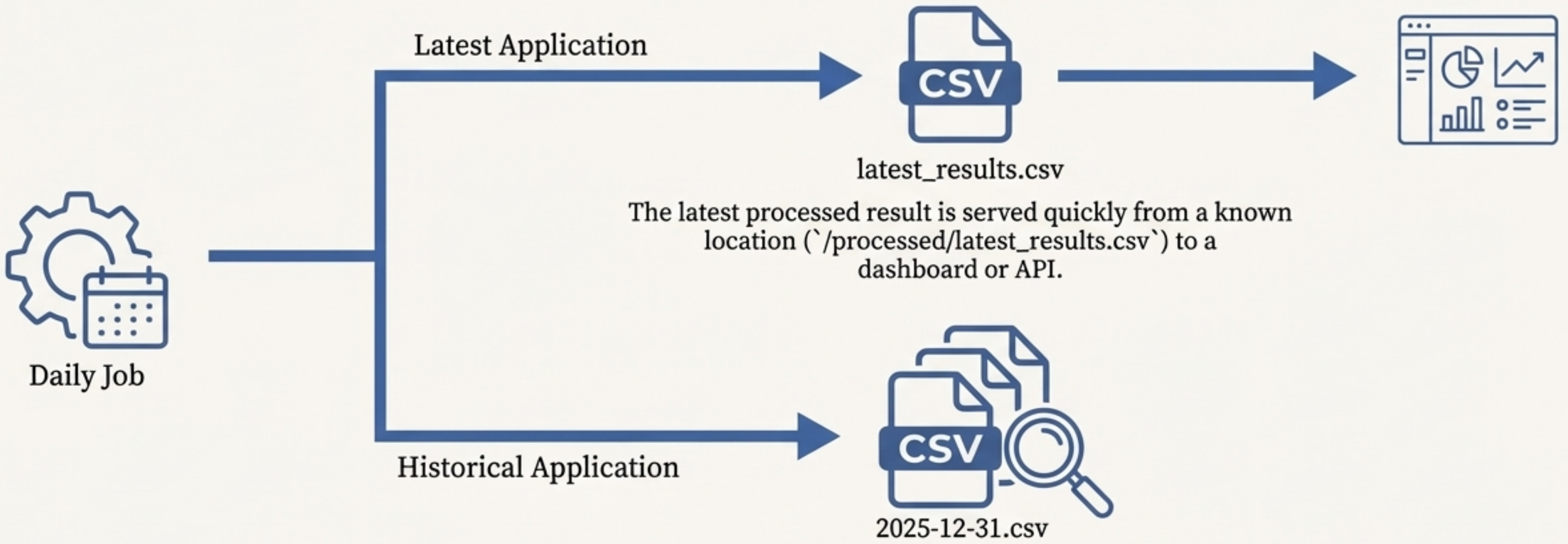
`14:32, yesterday:`
Port 8080 was closed.

`09:15, last week:`
Port 22 was restricted to a new IP range.

This makes it easy to pinpoint exactly when a security change was made.

Use Case: Data Pipeline Snapshots and Caching

Scenario: A daily web scraping job that processes and stores a key dataset.



latest_results.csv

The latest processed result is served quickly from a known location (`/processed/latest_results.csv`) to a dashboard or API.

2025-12-31.csv

Each daily run is also archived (`/processed/archive/2025-12-31.csv`). If a bug is found in the processing logic, engineers have a complete history of outputs to compare, debug, and recompute differences.

Use Case: Audit Logs and Compliance

Scenario: A system that manages user profiles, where changes must be logged.

users collection

```
Name: Alice Smith,
Address: 123 Main St
```

The main `users` collection always reflects the most current details for a user (e.g., their current address).

user_edits_log collection

```
[2023-10-27 14:32:01] Address changed
from '456 Oak Ave' by admin_user

[2023-10-26 09:15:45] Name changed
from 'Alice Jones' by alice_smith
```

Every single edit (address change, name update) is saved as an immutable document in a separate `user_edits_log` collection with a timestamp and the user who made the change.

Note: This naturally creates an audit log and aligns with established database patterns like MongoDB's Document Versioning Pattern.

A Pragmatist’s Guide: Implementation Considerations (Part 1)

Consistency of Dual Writes

Challenge: How do you avoid updating the 'latest' but failing to write the 'history', or vice-versa?

Solutions: Use database transactions where available. If not, establish a clear order of operations (e.g., write to history first, then update the 'latest' pointer).

Timestamp Granularity

Challenge: Updates can occur multiple times a second. How do you avoid collisions?

Solutions: Use a high-precision format like an ISO 8601 datetime string with milliseconds, or a simple incrementing version number.

History Pruning

Challenge: Historical data can grow very large over time.

Solutions: Define a retention policy. Implement strategies to prune or compress older versions (e.g., keep hourly snapshots for a week, then daily for a month).

A Pragmatist’s Guide: Implementation Considerations (Part 2)

Indexing for History

Challenge: How do you efficiently query the historical archive?

Solutions: If using a database, index key fields like timestamp or version ID for fast retrieval of date ranges or the last N versions.

Security & Access Control

Challenge: Historical data may contain sensitive information that has since been removed from the ‘latest’ record.

Solutions: Ensure access control policies apply to both latest and historical stores. Enforce rules about who can view archived data.

System Integrations

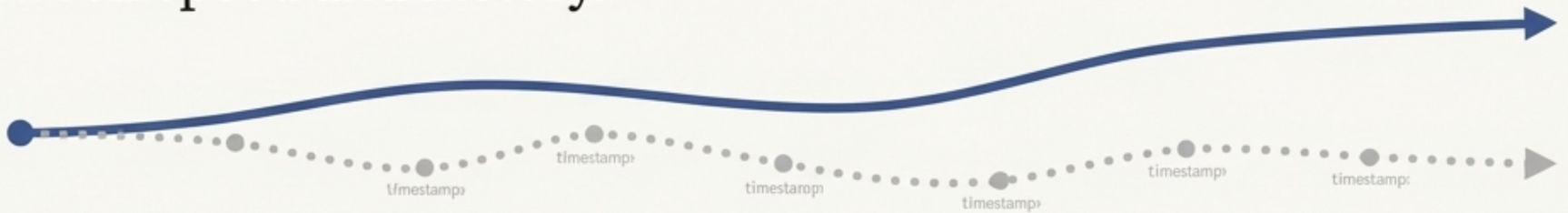
Challenge: Does your platform already offer this?

Solutions: Be aware of built-in features like temporal tables in relational databases or established patterns in NoSQL stores that can handle the dual-write logic for you.

The Dual Save Pattern: The Best of Both Worlds

[redacted]

By maintaining two copies of data—one as the ever-updating *latest*, and another preserving each change with a timestamp—the pattern elegantly resolves the conflict between speed and history.



Know what your data is now, an everything it has ever been.

Further Reading
- Data Versioning Explained: Guide, Examples & Best Practices (lakefs.io)
- Keep a History of Document Versions - MongoDB Docs
- Data Provenance in Healthcare - PMC