SITAC

STABLE-IDENTITY TEMPORAL AUDIT CHAIN

A database specification for complete, attributable temporal histories without changing the canonical identity of live records

FULL HISTORY. STABLE IDENTITY.

AUTHOR	**Paul Emerton**
PUBLISHER	**SPAWN.LONDON**
VERSION	**White Paper 1.0**
DATE	**17 August 2026**

CONTENTS

White Paper, Version 1.0

Abstract

Modern applications increasingly need to answer questions that ordinary current-state databases cannot answer on their own. What did this record contain before it was changed? Who created each version? When did that version become effective? Who superseded it? Was the record deleted, and was it subsequently restored?

Application and security logs remain essential, but they are often a poor primary mechanism for reconstructing the exact history of business data. Relevant events may be dispersed across services, retained for different periods, recorded as partial field changes, or mixed with large volumes of operational information. A technically complete answer can therefore require an investigator to correlate logs, database backups and application behaviour before the former state of one record can be established.

SITAC, the Stable-Identity Temporal Audit Chain, is an application-level temporal data specification developed by Paul Emerton. It uses copy-before-write snapshots while preserving one permanent canonical UUID for the live logical entity. Before an edit is applied, the complete current state is copied into a historical row with its own UUID. The canonical row is then updated in place and linked backwards to that snapshot. Each snapshot links to the version that preceded it.

The resulting chain makes the complete retained history of a logical record directly queryable from the record itself. Existing foreign keys continue to reference the same canonical UUID, regardless of how many times the record is edited, soft deleted or restored. This avoids repointing relationships and allows SITAC to be introduced into many database-centric applications without replacing their principal relational identity model.

SITAC has been deployed within ForgeBOS, an integrated Business Operating System in which customer, project, invoicing, scheduling, HR and operational records form a highly connected data model. This paper defines the method, its invariants, lifecycle operations, security controls, limitations and relationship to ISO/IEC 27001-aligned engineering.

Executive summary

SITAC is based on a simple rule:

> *The permanent identity of a live record should not change merely because the record has been edited.*

It achieves this through six core behaviours:

1. Every logical entity has one permanent canonical UUID.
2. The canonical row remains the row referenced by the rest of the application.
3. Before an edit, the current row is copied into a new historical snapshot.
4. The original canonical row is updated in place and points to the snapshot through `originating_id`.
5. Every earlier snapshot points to the version that preceded it, creating a reverse-linked temporal chain.
6. Soft deletion and restoration operate without changing the canonical UUID.

The practical result is:

- no foreign-key repointing when a record changes;
- no need to reconstruct business-data history from application logs;
- attributable version history using trusted user and timestamp metadata;
- direct point-in-time reconstruction;
- soft deletion with auditable restoration;
- controlled archival while retaining chain resolvability; and
- a clear architectural rule that can be supplied to human developers and AI coding systems.

SITAC is not a replacement for security logging, monitoring, access control, backups, retention policies or incident response. It is also not inherently tamper-proof. It provides an operational data history. Independent, protected backups or immutable archives provide stronger evidence if the operational environment itself is compromised.

1. The problem SITAC addresses

A conventional relational application usually stores the current state of an entity in one row. When the row is updated, the previous values are overwritten. The database remains operationally correct, but its former state no longer exists in the live data.

Many applications attempt to compensate through application logs. A log might state that a user changed a customer, amended an invoice, altered a project status or deleted an address. That can be useful evidence, but it does not always preserve the entire record as it existed before the change. It may record only the action, only the changed fields, or only a human-readable message generated by one version of the application.

The difficulty becomes greater as systems expand. Logs may be distributed between an API, worker processes, database servers, authentication services and infrastructure components. They may use different clocks, retention periods, identifiers and levels of detail. Investigating one disputed record can require a chain of inference rather than a direct data query.

The second problem is identity. In a mature relational application, a single record may be referenced from many places. A client organisation can be related to contacts, projects, quotations, invoices, schedules, messages, permissions, files and reporting data. Replacing the live record with a newly identified version can require every dependent relationship to be repointed, or can force the application to introduce another layer of identity indirection.

SITAC treats these as one design problem:

> *How can an application preserve every material state of a record while leaving the identity used by the rest of the database completely stable?*

2. Foundations and prior art

2.1 Temporal data

Temporal data architecture is an established field. It concerns information that changes over time and the ability to represent or query the periods during which particular facts or database states were valid.

System-time history records when a state existed in the database. Application-time history records when a fact was considered valid in the real world. A system that records both is commonly described as bitemporal. SITAC is principally a system-time pattern, although application-time fields can be added separately where the business domain requires them.

2.2 System-versioned temporal tables

Database platforms can provide native temporal features. Microsoft SQL Server system-versioned temporal tables, for example, keep a current table and a history table while the database engine manages the validity period of each row. This supports full data history and point-in-time analysis. Other platforms expose different temporal primitives. PostgreSQL 18, for example, provides temporal constraints over ranges for primary keys, unique constraints and foreign keys, rather than SQL Server's automatic current/history table model. [3] [4]

SITAC shares the objective of retaining previous states, but differs in its implementation and identity rules. It is application-managed, explicitly actor-attributed and centred on a permanent canonical UUID with a navigable predecessor chain. It can therefore be specified consistently across database platforms, including platforms that do not provide equivalent native temporal behaviour.

2.3 Event sourcing

Event sourcing records each domain change as an event in an append-only event store. The event stream is the source of truth, and current state is commonly derived or materialised from those events. [5]

SITAC is not event sourcing. Under SITAC, the canonical relational row remains the source of current truth. Historical snapshots preserve complete former states. The application does not normally need to replay every event to reconstruct the current entity.

2.4 Application and security logs

Logs answer questions about system and user activity. They may record authentication, rejected authorisation, administrative activity, exceptions, data exports, configuration changes and requests. OWASP recommends that logging mechanisms and stored log data be protected against unauthorised access, modification and deletion. [7]

SITAC answers a different question: what was the exact retained state of this business record at a particular point in time?

The two controls are complementary. SITAC should reduce dependence on logs for reconstructing data state, but it must not remove security logging or monitoring.

2.5 The distinct SITAC contribution

The broad concepts of temporal data, versioned records and copy-before-write history are established. SITAC does not claim to have invented those concepts.

Its distinct contribution is the formal combination of:

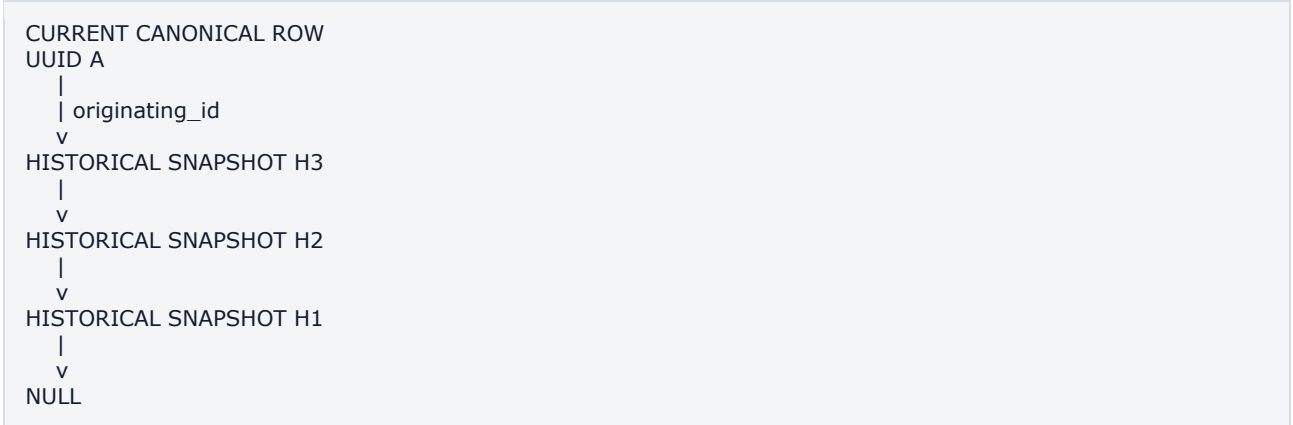
- one permanent canonical UUID for the logical entity;
- an unchanged live row that continues to satisfy existing foreign-key relationships;
- complete pre-mutation snapshots with their own UUIDs;
- a reverse-linked version chain using `originating_id`;
- attributable version start and end metadata;
- soft deletion represented on the canonical record;
- restoration that preserves the deleted interval;
- controlled archival without losing historical identity; and
- a database-independent specification suitable for human and AI-led implementation.

3. Formal definition

Stable-Identity Temporal Audit Chain, or SITAC, is an application-managed system-time temporal versioning pattern in which the permanent UUID of a logical entity remains bound to one canonical row, while every superseded state is copied into an immutable historical snapshot with a new UUID and linked as the immediate predecessor of the newer version.

The canonical row is the chain head. The rest of the application continues to reference it throughout the retained lifecycle of the entity.

A typical chain is:



Relationships elsewhere in the application remain directed at A:

```
Contacts ---------\
Projects ----------\
Invoices -----------+----> Canonical UUID A
Messages ----------/
Permissions -------/

Canonical UUID A ----> H3 ----> H2 ----> H1 ----> NULL
```

The relational identity and the temporal history are therefore separated without requiring a separate logical identity layer.

4. Normative design principles

The words **MUST**, **SHOULD** and **MAY** in this paper indicate mandatory, recommended and optional requirements respectively.

4.1 Permanent canonical identity

Every SITAC-controlled logical entity MUST have one canonical row and one permanent canonical UUID.

The UUID MUST remain unchanged through:

- creation;
- any number of edits;
- soft deletion;
- restoration; and
- archival of historical versions.

Ordinary foreign keys MUST continue to reference the canonical UUID. They MUST NOT be repointed merely because the canonical record has been edited.

4.2 Copy before mutation

Before any material edit is applied to the canonical row, the complete pre-edit state MUST be copied into a historical snapshot.

The snapshot MUST be committed in the same transaction as the canonical update. If either operation fails, both MUST roll back.

4.3 Complete predecessor chain

The canonical row MUST point to its immediate predecessor. Each historical snapshot MUST point to the version that preceded it.

The chain MUST be acyclic and SHOULD be non-branching. A historical snapshot should have no more than one successor.

4.4 Attributable version intervals

Each version MUST record:

- the trusted time at which it became effective;
- the authenticated actor responsible for creating that version;
- the trusted time at which it ceased to be effective, where applicable; and
- the authenticated actor responsible for superseding or deleting it.

4.5 Historical immutability

After a historical snapshot has been committed, ordinary application roles MUST NOT modify or physically delete it.

Any exceptional correction or purge MUST use a separately authorised and independently audited process.

4.6 Server-controlled metadata

Audit metadata MUST be derived from trusted server-side identity and time sources. Client-supplied JSON, form fields or API parameters MUST NOT be trusted to establish or overwrite SITAC metadata.

4.7 Soft deletion

Ordinary application deletion MUST be a soft deletion of the canonical row. The canonical UUID and business data MUST remain available to authorised history, restoration and retention processes.

4.8 Separate security evidence

SITAC MUST NOT be treated as a substitute for security logging, monitoring, backups, access control or incident response.

5. Core data model

A SITAC-controlled record contains the normal business fields plus the following mandatory metadata:

Field	Type	Null	Purpose
id	UUID	No	Physical row identity. On the canonical row, also the permanent logical identity.
created_by	UUID	No	Actor who created the current version represented by the row.
created_at	UTC timestamp	No	Time at which this version became effective.
deleted_by	UUID	Yes	Actor who superseded this historical version, or soft deleted the canonical entity.
deleted_at	UTC timestamp	Yes	Time at which this version ceased to be effective, or the canonical entity was soft deleted.
originating_id	UUID	Yes	UUID of the immediately preceding version.

A generic definition is:

```
id              UUID PRIMARY KEY,
created_by      UUID NOT NULL,
created_at      TIMESTAMP NOT NULL,
deleted_by      UUID NULL,
deleted_at      TIMESTAMP NULL,
originating_id  UUID NULL
```

5.1 Meaning of `id`

Every physical row has its own UUID.

For the canonical row, id is also the permanent logical identity used by APIs, routes and foreign keys.

For a historical snapshot, id identifies that specific former version. Historical snapshot IDs are not normally used as ordinary business relationships.

UUIDs are 128-bit identifiers designed for uniqueness without central registration and are widely used as database keys. RFC 9562 also notes the relevance of UUIDs to distributed database applications and defines time-ordered UUIDv7 alongside random UUIDv4. [6]

A UUID is an identifier, not an authorisation control. Every request must still enforce object-level and tenant-level authorisation. OWASP notes that UUIDs can still be involved in broken object-level authorisation if permission checks are absent. [12]

5.2 Meaning of `created_at` and `created_by`

Within strict SITAC semantics, created_at means the start of the row version's effective period. It does not necessarily mean the first creation date of the logical entity.

After an edit, the canonical row receives the edit timestamp and editing actor because it now represents a new current version. The original creation timestamp and actor remain preserved on the oldest snapshot.

Where an application requires the lifetime creation date to remain directly available without traversing history, it SHOULD add immutable fields such as:

```
entity_created_at
entity_created_by
```

This is particularly important during retrospective implementation, because existing reports may assume that created_at never changes.

5.3 Meaning of `deleted_at` and `deleted_by`

On a historical snapshot, these fields close the version's effective period. They are therefore equivalent to superseded_at and superseded_by in business meaning.

On the canonical row, they represent soft deletion of the logical entity.

This dual use is intentional but must be understood. Administrative interfaces SHOULD label the fields according to context rather than displaying every historical snapshot as though it were a deleted business entity.

5.4 Meaning of `originating_id`

Within SITAC, originating_id is a backward pointer to the immediately preceding version.

Its normative meaning is therefore:

```
previous_version_id
```

The name originating_id is retained because it is part of the deployed SITAC specification. Implementations MUST document the direction of the relationship unambiguously.

5.5 Recommended companion metadata

Higher-assurance systems SHOULD consider additional metadata:

Field	Purpose
operation_id	Correlates all rows changed within one business operation.
change_reason	Stores an authorised reason or ticket reference.
schema_version	Identifies the schema under which the snapshot was written.
source_system	Identifies imports, integrations or automated services.

Field	Purpose
request_id	Correlates the change with application and security logs.

These fields strengthen investigation and aggregate reconstruction, but they are not required for the core chain.

6. Record lifecycle

6.1 Creation

When a new logical entity is created, the application MUST:

1. Generate a new canonical UUID.
2. Set created_at from a trusted UTC clock.
3. Set created_by from the authenticated server-side actor.
4. Set deleted_at and deleted_by to NULL.
5. Set originating_id to NULL.
6. Insert the canonical row.

Example:

```
A
business_data    = Version 1
created_at       = T0
created_by       = U0
deleted_at       = NULL
deleted_by       = NULL
originating_id   = NULL
```

6.2 Editing

Every material edit MUST use one atomic transaction.

The application MUST:

1. Begin a transaction.
2. Lock or version-check the canonical row.
3. Confirm that it is not soft deleted.
4. Obtain one trusted UTC operation timestamp.
5. Obtain the authenticated actor UUID.
6. Generate a new snapshot UUID.
7. Copy the complete canonical row before applying changes.
8. Replace the copied row's id with the snapshot UUID.
9. Preserve the former created_at and created_by.
10. Copy the former originating_id into the snapshot.
11. Set the snapshot's deleted_at and deleted_by to the operation timestamp and actor.
12. Insert the snapshot.
13. Apply the requested business changes to the existing canonical row.
14. Leave the canonical id unchanged.
15. Set the canonical created_at and created_by to the operation timestamp and actor.
16. Set the canonical originating_id to the new snapshot UUID.
17. Ensure the canonical deleted_at and deleted_by remain NULL.
18. Commit the transaction.

Generic pseudocode:

```
BEGIN;

SELECT *
FROM entity
WHERE id = :canonical_id
FOR UPDATE;

INSERT INTO entity (
    id,
    business_fields,
    created_by,
    created_at,
    deleted_by,
    deleted_at,
    originating_id
)
SELECT
    :snapshot_id,
    business_fields,
    created_by,
    created_at,
    :actor_id,
    :operation_time,
    originating_id
FROM entity
WHERE id = :canonical_id;

UPDATE entity
SET
    business_fields = :new_values,
    created_by = :actor_id,
    created_at = :operation_time,
    deleted_by = NULL,
    deleted_at = NULL,
    originating_id = :snapshot_id
WHERE id = :canonical_id
  AND deleted_at IS NULL;

COMMIT;
```

The same operation timestamp SHOULD close the former version and open the new version. This creates contiguous half-open validity intervals:

```
[created_at, deleted_at)
```

6.3 Chain after repeated edits

After creation at T0, an edit at T1 and another edit at T2:

```
A: canonical current row
    data          = Version 3
    created_at    = T2
    created_by    = U2
    deleted_at    = NULL
    deleted_by    = NULL
    originating_id = H2

H2: historical snapshot
    data          = Version 2
    created_at    = T1
    created_by    = U1
    deleted_at    = T2
    deleted_by    = U2
    originating_id = H1

H1: historical snapshot
```

```
data          = Version 1
created_at    = T0
created_by    = U0
deleted_at    = T1
deleted_by    = U1
originating_id = NULL
```

The chain is:

```
A -> H2 -> H1 -> NULL
```

The rest of the database continues to reference A.

6.4 No-operation updates

An attempted update that does not change any persisted business value SHOULD NOT create a snapshot.

Where the attempted action itself is security-relevant, it SHOULD be recorded in the application security log rather than represented as a false data version.

6.5 Soft deletion

To delete a canonical entity, the application MUST:

1. Begin a transaction.
2. Lock the canonical row.
3. Confirm that it is not already deleted.
4. Set deleted_at to the trusted UTC deletion time.
5. Set deleted_by to the authenticated actor.
6. Leave the UUID, business data, creation metadata and predecessor link unchanged.
7. Commit the transaction.

A soft-deleted canonical row is still the chain head. It represents the final active version closed at the deletion time.

Ordinary active-data queries MUST exclude it.

6.6 Restoration

Restoration MUST preserve the evidence that the entity was absent for a period.

The application MUST:

1. Begin a transaction.
2. Lock the soft-deleted canonical row.
3. Generate a new historical snapshot UUID.
4. Copy the closed canonical version into the snapshot.
5. Preserve its former created_at, created_by, deleted_at, deleted_by and originating_id.
6. Insert the snapshot.
7. Clear deleted_at and deleted_by on the canonical row.
8. Set canonical created_at and created_by to the restoration time and actor.
9. Set canonical originating_id to the new snapshot UUID.
10. Commit the transaction.

The gap between the snapshot's deleted_at and the restored canonical row's new created_at is the interval during which the entity was deleted.

The canonical UUID remains unchanged.

6.7 Concurrency

Two simultaneous edits must not create a branched chain or silently overwrite one another.

SITAC therefore requires:

- a database transaction;
- row-level locking or equivalent optimistic concurrency control;
- verification that the expected chain head has not changed;
- rollback of both snapshot and canonical update on failure; and
- retry or conflict handling at the application layer.

7. Querying SITAC history

7.1 Current active records

Because every historical snapshot has a closing timestamp, the normal active-record filter is:

```
WHERE deleted_at IS NULL
```

This returns active canonical rows and excludes both historical snapshots and soft-deleted canonical rows.

Applications should enforce this through repositories, views, row scopes or another central persistence mechanism rather than relying on every developer to remember the filter.

7.2 Complete chain for one entity

Given the canonical UUID, a recursive query can follow the predecessor chain:

```
WITH RECURSIVE record_history AS (
   SELECT e.*, 0 AS depth
   FROM entity e
   WHERE e.id = :canonical_id

   UNION ALL

   SELECT previous.*, current.depth + 1
   FROM entity previous
   JOIN record_history current
     ON previous.id = current.originating_id
)
SELECT *
FROM record_history
ORDER BY depth ASC;
```

The canonical row is returned first, followed by progressively older versions.

7.3 Point-in-time reconstruction

Within one chain, a version was effective at time T where:

```
created_at <= T
AND
(
   deleted_at > T
   OR deleted_at IS NULL
)
```

If no version covers the requested time, the entity did not exist as an active record at that point, or the relevant history has passed its retention period.

7.4 Aggregate reconstruction

A record chain does not automatically reconstruct every related object.

Where a complete project, invoice, client or workflow aggregate must be reconstructed at one point in time:

- each mutable related record SHOULD implement SITAC;
- junction and relationship records SHOULD be versioned where their history matters;
- one `operation_id` SHOULD correlate all rows changed by the same business action; and
- participating records SHOULD use the same transaction timestamp where practical.

8. ForgeBOS implementation example

ForgeBOS is a Business Operating System developed for professional services and project-based businesses. Its public product material describes an integrated platform covering CRM, project management, invoicing, HR, email, scheduling and financial reporting, with deployment on customer infrastructure, in a managed data centre or through a chosen cloud provider. [13] [14]

This makes ForgeBOS a useful real-world implementation context for SITAC. A Business Operating System has a densely connected relational model. A client organisation may be referenced by contacts, opportunities, quotations, projects, tasks, communications, invoices, schedules, permissions and reports.

SITAC has been deployed within ForgeBOS so that these core entities retain stable canonical UUIDs while their former states remain directly available as temporal history.

8.1 Illustrative ForgeBOS client record

Consider a fictional ForgeBOS client organisation with canonical UUID:

```
018F2A34-7C11-7D90-9A31-0F82C0A91E55
```

The same UUID may be referenced by:

```
contact.organisation_id
opportunity.organisation_id
project.client_id
invoice.client_id
message.organisation_id
schedule.organisation_id
```

The organisation's address is then changed.

Under a version-replacement design, the application might create a new organisation identity and then need to repoint some or all of these relationships.

Under SITAC:

1. ForgeBOS copies the complete current organisation row into a historical snapshot with a new UUID.
2. The snapshot records when and by whom the former version ceased to be effective.
3. The original organisation row is updated in place.
4. Its permanent UUID remains `018F2A34-7C11-7D90-9A31-0F82C0A91E55`.
5. Its `originating_id` points to the snapshot.
6. Every contact, project, invoice and message remains correctly related without alteration.

If the organisation is edited again, another snapshot is inserted between the canonical row and the prior history.

If the organisation is deleted, ForgeBOS closes the canonical version through `deleted_at` and `deleted_by`. If it is later restored, the closed state is copied into history and the same canonical UUID is reopened as the current version.

8.2 What ForgeBOS can answer directly

An authorised ForgeBOS user or auditor can retrieve the record chain and determine:

- the organisation's original retained data;
- every retained material version;
- the effective period of each version;
- the actor responsible for each new version;
- the actor whose action closed each previous version;
- the state immediately before deletion;

- the duration of any deleted interval; and
- the current canonical state.

The data history does not need to be rebuilt by correlating application logs. Logs can instead be used for their proper purposes, such as confirming authentication context, authorisation decisions, request origin, administrative activity and security events.

8.3 Why stable identity matters particularly in ForgeBOS

The value of SITAC increases with relational density.

In a small standalone table, replacing a record identity may be manageable. In ForgeBOS, the same organisation, project or user can participate in many operational processes. Stable canonical identity means that temporal versioning can be added without turning every edit into a relationship migration.

This is also important for integrations. External systems can retain the canonical UUID as a durable reference while ForgeBOS continues to maintain its internal history.

9. Retrospective implementation

SITAC can often be introduced into an established database-centric application without replacing existing foreign keys, because the existing live row remains the canonical row.

That does not mean implementation is automatic. The mutation path, query model, constraints and retention controls must still be reviewed.

9.1 Recommended migration sequence

1. **Classify entities.** Identify which tables contain material mutable business data and which are transient, derived or high-frequency operational data.
2. **Confirm canonical identity.** Ensure each in-scope entity has a permanent UUID. Where a legacy integer exists, introduce a UUID as the authoritative application identity before versioning.
3. **Add SITAC metadata.** Add `created_by`, `created_at`, `deleted_by`, `deleted_at` and `originating_id` using an approved migration.
4. **Establish a baseline.** Treat the existing live row as the first known canonical state. Where historical actor or time evidence is unavailable, use a documented migration actor and do not invent provenance.
5. **Centralise mutations.** Route edits, deletes and restorations through a transactionally enforced persistence service, stored procedure or trigger.
6. **Protect metadata.** Reject client attempts to set audit fields directly.
7. **Filter operational reads.** Ensure ordinary reads exclude historical and soft-deleted rows.
8. **Review uniqueness.** Adjust unique business constraints so snapshots do not conflict with the current row.
9. **Add history queries.** Provide an authorised history service or administrative interface.
10. **Test concurrency and rollback.** Prove that the chain cannot branch or partially commit.
11. **Implement retention.** Define archival, legal hold and disposal procedures.
12. **Validate backups.** Test recovery and compare recovered chains with operational expectations.

9.2 Principal application logic that can remain unchanged

Where the canonical UUID already exists and the application reads current records through a central data layer:

- inbound foreign keys can remain unchanged;
- public routes and API object references can remain unchanged;
- current-state business logic can continue reading the canonical row;
- integrations can continue using the same durable identity; and
- reports that consume current data can often remain unchanged.

9.3 Areas that require review

Retrospective deployment must still assess:

- every update and delete route;
- bulk imports and maintenance scripts;
- direct database access;
- unique indexes on copied business fields;
- assumptions that `created_at` never changes;
- storage growth from full-row snapshots;
- large objects and attachments;
- tenant isolation;
- history permissions;
- schema evolution; and
- data retention and erasure.

The claim is therefore not that SITAC requires no application work. Its benefit is that it avoids a much larger class of work: replacing or repointing the established identity graph.

10. Database constraints and performance

10.1 Recommended indexes

A SITAC implementation SHOULD normally include:

- a primary index on `id`;
- an index on `originating_id`;
- a uniqueness rule preventing more than one successor from referencing the same predecessor, where the database supports it;
- an index supporting `deleted_at IS NULL` current-record queries;
- indexes on `created_at` and `deleted_at` for temporal searches;
- actor indexes where audit reporting requires them; and
- tenant-scoped composite indexes in multi-tenant deployments.

10.2 Unique business fields

Copying a complete row can conflict with ordinary unique constraints. For example, a snapshot of a user could contain the same email address as the current canonical row.

The database design must therefore apply business uniqueness to active canonical records rather than every physical historical row. Depending on the database, this may use:

- filtered or partial unique indexes;
- generated columns;
- a separate history table;
- a canonical-only uniqueness registry; or
- another tested constraint strategy.

SITAC's identity and chain semantics do not require historical rows to share the same physical table, provided every historical UUID remains resolvable and the copy-before-write lifecycle remains atomic.

10.3 Write and storage amplification

Each material edit normally produces one snapshot insert plus one canonical update. Storage grows with the number of versions and the size of each row.

SITAC is therefore well suited to material business records but may be unsuitable for:

- high-frequency telemetry;
- rapidly changing counters;
- cache entries;
- transient session data;
- continuously updated position data; or
- large binary values copied on every edit.

Large attachments SHOULD normally be stored separately and referenced through versioned metadata, rather than repeatedly copied into every row snapshot.

10.4 History growth and archival

Database vendors warn that temporal history can increase database size significantly where retention is long or update frequency is high. [3]

SITAC implementations SHOULD monitor:

- versions per entity;
- history growth per table;
- snapshot insert latency;
- recursive query depth;
- archive volume;
- broken or unresolved predecessor links; and
- retention exceptions.

11. Archival and retention

Historical snapshots MAY be moved from operational storage into:

- archive tables;
- partitioned history stores;
- structured JSON archives;
- immutable object storage; or
- independently controlled cold storage.

An archive record MUST preserve at least:

```
snapshot id
canonical id or resolvable chain ownership
entity type
business data
created_by
created_at
deleted_by
deleted_at
originating_id
schema version
archive timestamp
archive batch id
```

Before the operational copy is removed, the archive process MUST verify that:

1. the exported record is complete;
2. the historical UUID remains retrievable;
3. the predecessor link remains resolvable;
4. the archive can be read and restored;

5. the applicable retention policy permits the move; and
6. the archive security boundary is appropriate.

SITAC does not justify indefinite retention. The ICO's storage-limitation guidance states that personal data should not be kept longer than necessary, that retention periods should be justified and documented, and that data should be erased or anonymised when it is no longer required. Moving personal data offline does not remove it from data-protection obligations. [11]

A system may only claim a complete history for the period that it actually retains. After authorised permanent disposal, it should describe the result as a complete retained history within the documented retention period.

12. Security and tamper resistance

12.1 Threat model

SITAC provides strong operational traceability when application users and ordinary services cannot modify committed historical snapshots.

It does not automatically protect against an attacker or administrator who has sufficient privilege to rewrite the database, alter audit metadata and modify the chain itself.

This distinction must be explicit:

```
SITAC chain integrity
    is not the same as
independent proof that the chain was never rewritten
```

12.2 CRC values

A cyclic redundancy check is appropriate for detecting accidental corruption in many contexts, but it is not a cryptographic hash. NIST describes CRC as a checksum used where accidental changes are expected. [10]

A CRC stored with a SITAC row should therefore not be represented as protection against a malicious history rewrite. An attacker who can alter the row can usually calculate a replacement CRC.

12.3 Cryptographic hash chains

A cryptographic predecessor hash can make unauthorised modification easier to detect where the attacker cannot replace every later hash and any trusted anchor.

For example:

```
hash(H1)
hash(H2 + hash(H1))
hash(H3 + hash(H2))
```

However, if a privileged attacker controls both the history and the stored hashes, the attacker may rewrite the history and recalculate the chain.

A hash chain becomes materially stronger when its head or periodic digest is anchored outside the compromised trust boundary.

12.4 Independent backups and immutable storage

For most business applications, independently secured backup generations can provide more practical compromise recovery and forensic comparison than a CRC attached to every row.

The backup environment SHOULD have:

- credentials separate from the production application;
- restricted deletion and overwrite capability;

- multiple retained generations;
- encryption and key management;
- tested restoration procedures;
- monitoring of backup failures and administrative actions; and
- periodic offline or immutable copies where justified by risk.

NIST data-integrity guidance treats backups, secure storage, integrity monitoring and logging as complementary capabilities. Secure storage is intended to prevent alteration of protected files, while backups provide recovery points created before a destructive event. [9]

An optional low-overhead enhancement is a periodic cryptographic manifest of an export or backup, stored with the independently protected copy. This creates an external reference without requiring every operational record to perform a cryptographic chain calculation on every edit.

12.5 Security logging remains mandatory

SITAC records the state history of business data. Security logs should continue to record matters such as:

- authentication success and failure;
- authorisation failure;
- administrative and privileged actions;
- security configuration changes;
- unusual data access;
- exports and bulk downloads;
- direct database maintenance;
- archive and purge operations;
- exceptions and failed transactions; and
- attempts to bypass the SITAC persistence path.

OWASP specifically recommends protecting logs from tampering and ensuring transactions have audit trails with integrity controls. [7] [8]

12.6 Access control

The history interface may expose sensitive former values that are no longer visible in the current row. Access to history therefore requires its own authorisation rules.

A user authorised to view the present address, salary, project status or contact information is not automatically authorised to see every former value.

Multi-tenant implementations MUST ensure that every chain node belongs to the same tenant as the canonical row and that chain traversal cannot cross tenant boundaries.

13. ISO/IEC 27001 alignment

ISO/IEC 27001 defines requirements for establishing, implementing, maintaining and continually improving an Information Security Management System, or ISMS. It uses a risk-based, organisation-wide approach covering people, policies and technology. Certification applies to an organisation's scoped management system, not to one database pattern. [1]

ISO/IEC 27002 provides guidance and control objectives that organisations can apply within an ISMS. It is guidance rather than a separately certifiable standard. [2]

SITAC can contribute technical evidence and control operation in areas concerning:

- collection and preservation of evidence;
- protection of records;
- information deletion;

- logging and monitoring;
- clock synchronisation;
- secure development lifecycle;
- application security requirements;
- secure architecture and engineering principles;
- secure coding; and
- change management.

Potentially relevant ISO/IEC 27001:2022 Annex A controls include 5.28, 5.33, 8.10, 8.15, 8.16, 8.17, 8.25, 8.26, 8.27, 8.28 and 8.32. Applicability must be determined through the organisation's risk assessment and Statement of Applicability, and control names and requirements should be verified against the organisation's licensed copy of the standards.

13.1 What SITAC can evidence

A correctly implemented SITAC chain can demonstrate:

- that previous retained states were preserved;
- which authenticated actor created each retained version;
- when each version became and ceased to be effective;
- that the canonical identity remained stable;
- that deletion and restoration were attributable;
- that ordinary application users could not silently overwrite history; and
- that archived histories remained linked to their canonical entities.

13.2 What SITAC cannot evidence alone

SITAC alone does not prove:

- that a privileged administrator never rewrote the database;
- that an actor's credentials were not compromised;
- that access to data was authorised;
- that every security event was logged;
- that backups are usable;
- that retention periods are lawful;
- that incident response procedures work;
- that the organisation has an effective ISMS; or
- that the organisation is certified to ISO/IEC 27001.

The appropriate public statement is therefore:

> *SITAC is designed to support ISO/IEC 27001-aligned requirements for auditability, traceability, record protection and secure application development.*

It should not be described as independently ISO 27001 compliant or certified.

14. Conformance requirements

An implementation should not be described as SITAC-conformant unless it passes at least the following tests.

14.1 Stable identity test

After repeated edits, deletion and restoration, the canonical UUID remains exactly the UUID assigned at initial creation.

14.2 Snapshot completeness test

Every successful material edit produces exactly one historical snapshot containing the complete pre-edit business state.

14.3 Chain test

The canonical row links to the immediate predecessor and every predecessor link resolves without cycles or unexpected branches until NULL.

14.4 Temporal boundary test

For an edit, the previous version's deleted_at equals the newer version's created_at.

14.5 Actor attribution test

For an edit, the previous version's deleted_by and the newer version's created_by identify the authenticated editing actor.

14.6 Foreign-key stability test

No inbound business foreign key changes merely because the canonical record is edited.

14.7 Atomic rollback test

If either the snapshot insert or canonical update fails, neither change remains committed.

14.8 Concurrent-edit test

Two concurrent edits cannot silently overwrite each other or produce competing predecessors.

14.9 Soft-delete test

Deletion leaves the canonical UUID and business data present while ordinary active queries exclude the entity.

14.10 Restoration test

Restoration retains the canonical UUID, preserves the closed pre-deletion version and records a new current effective period.

14.11 Historical immutability test

Ordinary application roles cannot update or physically delete historical snapshots.

14.12 Archive resolution test

After archival, every retained historical UUID and predecessor link remains resolvable through the authorised history service.

15. SITAC within AI-accelerated software development

AI coding systems can implement a pattern quickly, but they will usually select conventional CRUD behaviour unless the required architectural invariants are made explicit.

A feature request such as "allow the user to edit and delete a client" does not tell an AI system that:

- the client UUID must remain permanent;
- the complete previous state must be retained;
- the former version must be actor-attributed;
- soft deletion is mandatory;
- restoration must preserve the deleted interval;

- historical rows are immutable;
- the operation must be atomic; or
- security logs and independent backups remain separate requirements.

SITAC is therefore recorded as a mandatory requirement within the **Core Software Design Requirements Specification**, or CSDRS.

This allows the same rule to ground:

- database design;
- API design;
- repository and service implementation;
- migration scripts;
- administrative tools;
- code review;
- automated testing; and
- AI-generated code.

The wider engineering principle is important:

> *As more software is generated from lists of desired features, core software design requirements provide the persistent rules that generated code must not compromise.*

16. Limitations and non-goals

SITAC is deliberately focused. It is not intended to solve every form of audit, security or temporal modelling.

It is not:

- a replacement for application or security logs;
- a substitute for backups or disaster recovery;
- inherently cryptographically tamper-proof;
- a non-repudiation system;
- a replacement for access control;
- an event-sourcing architecture;
- automatically bitemporal;
- suitable for every high-frequency data type;
- permission to retain personal data indefinitely; or
- proof of ISO/IEC 27001 compliance or certification.

A complete historical claim also depends on implementation discipline. If direct SQL updates, bulk scripts or integrations can bypass SITAC, the history may be incomplete even though the schema exists.

17. Conclusion

SITAC was developed to resolve a practical tension in auditable relational software.

Applications need to preserve how their data changed, but the identity used by the rest of the system should remain stable. Creating replacement identities for every edit can destabilise relationships. Relying on logs can make a simple historical question depend on a complex forensic reconstruction.

SITAC combines temporal snapshots with permanent canonical identity.

Before mutation, the current state is preserved. The original row remains current. Its UUID remains unchanged. Its predecessor link makes the complete retained history directly traversable. Soft deletion and restoration continue the same chain. Archival can reduce operational storage while retaining historical resolvability.

In ForgeBOS, this allows highly connected business entities to retain durable identities across CRM, projects, invoicing, scheduling and related operations while maintaining an attributable chain of change.

The method does not replace the rest of a secure architecture. Logs establish security activity. Access controls determine who may act. Independent backups and immutable storage strengthen recovery and evidential integrity. Retention policies determine how long history should exist. An ISMS governs the overall risk.

Within those boundaries, SITAC offers a clear and reusable design rule:

Full history. Stable identity.

Appendix A: Minimum AI implementation brief

The following may be copied into an AI coding brief:

```
This application uses SITAC, the Stable-Identity Temporal Audit Chain.

Every logical entity has one permanent canonical UUID. The canonical row and UUID remain the live record referenced by all existing foreign keys throughout creation, editing, soft deletion and restoration.

Before any material edit, copy the complete current canonical row into a new historical snapshot with a new UUID. Preserve the former created_at, created_by and originating_id. Set the snapshot's deleted_at and deleted_by to the trusted UTC edit time and authenticated editing actor.

Then update the existing canonical row in place. Do not change its UUID. Apply the business changes, set created_at and created_by to the same edit time and actor, clear deleted_at and deleted_by, and set originating_id to the newly created snapshot UUID.

The canonical row therefore points to its immediate predecessor. Every copied snapshot retains the predecessor pointer that existed before it was copied, creating a reverse-linked chain to the earliest version.

Perform the snapshot insert and canonical update within one atomic transaction using row locking or equivalent concurrency protection. Historical snapshots are immutable to ordinary application roles.

Deletion is a soft delete of the canonical row using deleted_at and deleted_by. Restoration first copies the closed canonical version into history, then clears deletion metadata on the same canonical UUID, creates a new current effective period and links the canonical row to the copied predecessor.

Audit metadata is server-controlled and must not be trusted from client input. All timestamps are UTC. UUIDs identify records but do not replace object-level or tenant-level authorisation.

SITAC does not replace application security logs, independent backups, retention controls or incident monitoring.
```

Appendix B: Recommended chain integrity checks

A scheduled integrity process SHOULD identify:

- originating_id values that do not resolve;
- self-referencing rows;
- cycles;
- one predecessor referenced by multiple successors;
- history rows with missing closing metadata;
- invalid timestamp order;
- actor UUIDs that cannot be resolved through the audit identity service;
- chain nodes crossing tenant boundaries;
- canonical edits without matching snapshots;
- archive records missing from the history resolver; and

- direct database mutations that bypass the approved persistence path.

References

1. International Organization for Standardization. ISO/IEC 27001:2022: Information security, cybersecurity and privacy protection - Information security management systems - Requirements.
2. International Organization for Standardization. ISO/IEC 27002:2022: Information security, cybersecurity and privacy protection - Information security controls.
3. Microsoft Learn. Temporal Tables - SQL Server.
4. PostgreSQL Documentation. CREATE TABLE: temporal constraints using WITHOUT OVERLAPS and PERIOD.
5. Microsoft Azure Architecture Center. Event Sourcing Pattern.
6. IETF RFC Editor. RFC 9562, Universally Unique IDentifiers (UUIDs), May 2024.
7. OWASP Cheat Sheet Series. Logging Cheat Sheet.
8. OWASP Top 10:2025. A09 Security Logging and Alerting Failures.
9. National Institute of Standards and Technology. Data Integrity practice guides, including NIST SP 1800-25 and NIST SP 1800-11.
10. NIST Computer Security Resource Center. Cyclic Redundancy Check (CRC) glossary entry.
11. Information Commissioner's Office. Principle (e): Storage limitation.
12. OWASP Cheat Sheet Series. Insecure Direct Object Reference Prevention and Multi-Tenant Security.
13. ForgeBOS product material.
14. SPAWN.LONDON. ForgeBOS case study.

About the author

Paul Emerton develops software, systems architecture and AI-accelerated applications through SPAWN.LONDON. His work focuses on turning commercial requirements into practical, governed systems, including the recovery and modernisation of established software and the definition of reusable core engineering requirements.

Publication

Published by SPAWN.LONDON AI-accelerated software and application development London-based, globally delivered

SPAWN.LONDON

AI-accelerated software and application development