

ClaritasZ

Provenance Framework v1.0

Complete Documentation

Making architecture decisions discoverable



Table of Contents

1. Framework Philosophy
2. Database Schema
3. Copilot Analysis Workflow
4. Amendment Ingestion
5. 5-Phase Implementation
6. Terms & Conditions
7. TOGAF Complementarity

Clarity7[®] JPS

1. Framework Philosophy

Why Amendments, ADRs, and Documents Are Structured This Way

****ClaritasZ | July 2026 | 2 Pages****

Why Amendments Come First

An amendment is an observation made explicit.

When you read an architecture document, you see claims: "We standardize on Kubernetes." "Security is cross-cutting." "Cloud-first for non-production." These claims are **already** decisions someone made. But they're buried in prose. No one knows who decided, when, or why.

An amendment says: this section of the document expresses a choice. We're making that choice visible as a unit. Not hidden in a paragraph. Not layered under other decisions. Just: here is what this part of the architecture establishes.

Amendments are the first layer because they're the closest thing to raw observation. They're what the document **says** — not our interpretation of what it means. They're what we can all agree on: "Yes, Chapter 3 does claim that." Even if we later disagree about whether the claim is wise.

By treating amendments as first-class objects, we do two things:

1. ****We make change trackable.**** Every revision to the document is an amendment. Who approved it? When? What changed from the previous version? All visible.
2. ****We force clarity.**** You can't amend something vague. "This section is unclear" is not an amendment. "This section should say X instead of Y" is. Amendments demand precision.

Why Architectural Decisions Come Later

An ADR is a claim verified.

After you've documented amendments — after you've said "here is what the architecture establishes" — you ask a second question: **Is this decision intentional?** Did someone actually decide this, or did it just happen?

A PROPOSED ADR starts as a hypothesis: "The document suggests we decided X." The Domain Architect reviews it and asks: "Is X a real decision we made, or is it just something the document happens to say?" If it's real, the ADR becomes ACCEPTED. If it's accidental, the ADR is SUPERSEDED — the document is amended to remove the accidental implication.

ADRs come later because they require judgment. An amendment is a fact about the document. An ADR is a claim about the architecture's intent. You can observe amendments. You must **verify** ADRs.

This sequencing has a crucial implication: ****the document is not the source of truth about decisions. The document is the artifact that results from decisions.**** The real decisions live in the ADRs. The document is derived from them.

Why Documents Are Derived Artifacts

A document is a snapshot.

Once you've captured amendments and verified the ADRs behind them, the document becomes **derived**. It's generated from the database: "Here are all the amendments we've approved. Here are the ADRs they implement. Render all of this as a Docx/HTML/Markdown."

This is a fundamental inversion from how architecture documents usually work:

****Old way:**** Write document → read document → try to extract decisions

****Provenance way:**** Make decisions explicit (amendments + ADRs) → generate document from them → document is always consistent with decisions

By making documents derived, we gain:

1. ****Consistency.**** You can't accidentally make the document and the decisions contradict. If the document is generated from the decisions, they're always in sync.
2. ****Traceability.**** Every sentence in the document can be traced back to an amendment. Every amendment can be traced back to an ADR (or left standalone). Change history is complete.
3. ****Provenance.**** The document doesn't hide its ancestry. You can ask: "Who approved this? When? Why?" The answer is in the database.

Documents stop being the primary artifact. They become **outputs**. The database — amendments and ADRs — becomes the source of truth.

How Architectural Thinking Evolves: From Observation to Decision

This framework reflects a journey that architecture must take:

****Phase 1: Observation****

You read the existing architecture. You make amendments. You're saying: "Here is what exists. Here is what the current document establishes." No judgment yet. Just clarity.

****Phase 2: Verification****

You look at those amendments and ask: "Which of these represent intentional decisions?" You propose ADRs. The Domain Architect reviews and decides: "Yes, this is intentional" (ACCEPT) or "No, this was accidental, we should fix it" (SUPERSEDE).

****Phase 3: Evolution****

Once decisions are explicit and verified, you can **evolve** them. You can propose a new amendment that supersedes an old one. You can create a new ADR that replaces an old one. Every evolution is traceable. Every old decision remains in the database. The architecture doesn't forget its past.

****Phase 4: Governance****

With decisions explicit and traceable, you can govern them. You can ask: "Do all our decisions still hold? Have circumstances changed? Should we revisit any of these?" You can audit. You can plan the transition from current state (IST) to desired state (SOLL). You can see dependencies: "If we change ADR-001, what amendments are affected?"

The evolution is from implicit to explicit, from undocumented to traceable, from brittle to resilient.

The Core Insight

****Amendments are observations. ADRs are verifications. Documents are artifacts of both.****

By separating these layers, you make architecture discussable. You can agree on what exists without agreeing on what to do about it. You can verify decisions without re-arguing the observations that led to them. You can evolve the architecture while keeping its history intact.

The Provenance Framework codifies this separation. It says: if you want to understand your architecture, you must first make it explicit at three levels:

1. ****What does the document say?**** (Amendments)
2. ****Which of those claims are intentional decisions?**** (ADRs)
3. ****What does this all look like when rendered?**** (Documents)

Get those three layers right, and architecture becomes manageable. Decisions become discoverable. Change becomes governable. The architecture can evolve without losing its mind.

****The philosophy in one sentence:**** Make observations explicit (amendments), verify them as decisions (ADRs), then derive everything else (documents, roadmaps, governance) from that foundation.

Clarity7[®]
JPS

2. Database Schema

```
-- Provenance Framework v1.0 – SQLite Schema
-- Generic domain architecture document management
-- with ADR tracking and amendment discipline
-- =====
-- NUMBERING SCHEMES
-- =====
-- ADR numbering: ADR-### (global sequential)
-- Example: ADR-001, ADR-002, ADR-003, ...
-- Auto-generated at insertion time
--
-- Amendment numbering: A-### (global sequential)
-- Example: A-001, A-002, A-003, ...
-- Auto-generated at insertion time
--
-- Document version numbering: customer-defined
-- Examples: v1.0, v1.1, Release1, 2026-Q3
-- No enforcement; customer chooses their scheme
-- =====
-- CORE TABLES
-- =====
CREATE TABLE identities (
identity_id TEXT PRIMARY KEY,
name TEXT NOT NULL,
email TEXT,
role TEXT,
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
CREATE TABLE documents (
document_id TEXT PRIMARY KEY,
document_name TEXT NOT NULL,
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
```

3. Copilot Analysis Workflow

Copilot Prompt for Initial Document Ingestion

Instructions for You (the user)

1. Copy this entire prompt
2. Paste it into Claude or your Copilot interface
3. Attach or paste your IST/SOLL domain architecture document
4. Run it
5. Copy the JSON output
6. Use it with the amendment ingestion guide

Prompt to Send to Copilot

You are a domain architecture analyst. Your task is to analyze a domain architecture document and extract its structure for version control and amendment tracking.

****Document Analysis Task:****

Analyze the attached document thoroughly. Your goal is to make implicit decisions explicit and identify all the foundational choices baked into the current architecture.

Extract the logical chapters and sections. For each chapter:

- ****Chapter name**** (as it appears)
- ****Sections within it**** (all subsections, no matter how deep)
- ****Brief description**** (one sentence: what is this chapter about?)

For each chapter, write a PROPOSED amendment that documents what this chapter establishes:

- ****Title****: What is the core claim of this chapter?
- ****Summary****: 2-3 sentences. What does this chapter define or decide?
- ****Rationale****: Why is this important? What would break if this chapter was wrong?
- ****Applies to****: Which chapter(s) and section(s) does this cover?

Think: if someone had to explain this chapter to someone who'd never read it, what would you say?

Look for decisions embedded in the text that are NOT explicitly called out as "decisions". Examples:

- "We use Kubernetes" (implies: cloud-native, orchestration choice)
- "Security is a cross-cutting concern" (implies: security governance model)
- "We standardize on [technology]" (implies: standardization choice, vendor lock-in acceptance)

For each implicit ADR you find:

- ****What decision is baked in?****
- ****Where in the document does this appear?****
- ****What would change if we reversed this decision?****

What major architectural topics are missing from this document that would be needed for:

- A complete SOLL (target) state?

- A transition/roadmap plan?
- Operational governance?
- Risk and dependency management?

Output Format

Return **only** valid JSON. No preamble, no explanation, just the JSON block.

```
```json
{
 "document_metadata": {
 "document_name": "[What the customer calls this document]",
 "analysis_date": "[Today's date]",
 "document_size_pages": "[Approx number of pages]"
 },
 "structure": [
 {
 "chapter_number": 1,
 "chapter_name": "[Exact title from document]",
 "chapter_description": "[One sentence about what this chapter covers]",
 "sections": [
 {
 "section_name": "[Exact name]",
 "section_path": "Chapter 1 → Section 1.1 → Subsection 1.1.1",
 "section_type": "intro | architectural | operational | governance"
 }
]
 }
],
 "proposed_amendments": [
 {
 "amendment_title": "[Core claim of this chapter]",
 "change_summary": "[2-3 sentence summary of what this chapter establishes]",
 "change_rationale": "[Why this is important for the architecture]",
 "applies_to": "[Chapter X, Sections Y and Z]",
 "amendment_type": "structural | governance | technical"
 }
],
 "implicit_adrs": [
 {
 "adr_title": "[What decision is baked in?]",

```

```

"where_found": "[Chapter X, Section Y — exact quote or location]",
"implications": "[What would change if we reversed this?]",
"criticality": "critical-foundational | important-operational | nice-to-document"
}
],
"identified_gaps": [
{
"gap_category": "SOLL target state | transition roadmap | governance | risk | other",
"gap_description": "[What's missing?]",
"why_matters": "[Why would this matter for the architecture?]"
}
]
}
...

```

### **Key Principles**

- **Be specific.** Don't say "the document discusses infrastructure" — say "Chapter 3 defines a layered model with compute, network, and storage as separate governance domains."
- **Surface implicit choices.** If the document says "we use cloud-first strategy," that's a decision. Capture it.
- **Look for tensions.** If the document says both "standardize everything" and "allow flexibility," note that tension.
- **Focus on architecture, not implementation.** We care about *decisions* and *structures*, not configuration details.
- **For large documents (100+ pages):** Analyze systematically per chapter. One amendment per chapter minimum. You can group closely related sections into a single amendment if they express one unified architectural idea.

---

### **Next Steps (After Analysis)**

1. **You review the JSON output** — Does it capture the document accurately? Do the amendments make sense?
2. **Send to your architecture team** — Share the JSON with your Domain Architect (e.g., Jaap). They will review each proposed amendment.
3. **Architect approves/rejects** — The Domain Architect reviews the amendments and either approves them (status = APPROVED) or asks for changes.
4. **Ingest into database** — Once approved, use the *amendment-ingestion-guide.pdf* to load the amendments into your Provenance database.
5. **Generate next version** — The database will generate the next document version with all approved amendments incorporated.

---

**Ready?** Paste this prompt + your document into Claude/Copilot and run it.

## 4. Amendment Ingestion

### *Loading Copilot Analysis into the Provenance Database*

---

#### **Overview**

After Copilot analyzes your document, you get a JSON file with:

- Document structure (chapters/sections)
- Proposed amendments
- Implicit ADRs
- Identified gaps

This guide walks you through:

1. **\*\*Prepare the JSON\*\*** — validate it
2. **\*\*Domain Architect review\*\*** — approve/reject each amendment
3. **\*\*Ingest into database\*\*** — load amendments with PROPOSED status
4. **\*\*Generate next version\*\*** — create document version from amendments

---

#### **Step 1: Validate the JSON Output**

After running the Copilot analysis prompt:

...

1. Copy the JSON output from Copilot
2. Save it as: analysis-.json
3. Validate it (use any JSON validator, e.g., jsonlint.com)
4. Check it's well-formed (no syntax errors)

...

If validation fails: Run the Copilot analysis again.

---

#### **Step 2: Domain Architect Review**

**\*\*You (the Domain Architect) now review the JSON.\*\***

For each `proposed\_amendment` in the JSON:

- [ ] **\*\*Does the summary capture what this chapter actually says?\*\***
- [ ] **\*\*Is the rationale compelling?\*\*** Why does this decision matter?
- [ ] **\*\*Does it apply to the right sections?\*\*** ("applies\_to" field)
- [ ] **\*\*Is anything missing?\*\*** Could this amendment be split into two?

**\*\*Accept as-is:\*\***

- The amendment is accurate and complete
- Status: APPROVED (you'll set this in Step 3)

**\*\*Request changes:\*\***

- Edit the JSON amendment directly
- Mark it with a note: `"status_note": "Domain Architect: needs clarification on X"`
- Re-send to Copilot or edit manually

**\*\*Reject:\*\***

- This amendment doesn't belong
- Remove it from the JSON entirely
- Note why in a comment file

---

### **Step 3: Ingest into Database**

Once you've reviewed all amendments:

```
```bash
```

```
sqlite3 provenance.db < schema.sql
```

```
sqlite3 provenance.db << 'EOF'
```

```
INSERT INTO identities (identity_id, name, email, role) VALUES
```

```
('domain-arch-001', '[Architect Name]', '[domain-architect@example.com]', 'Domain Architect'),
```

```
('copilot-000', 'Copilot', 'system@provenance', 'Analysis Tool');
```

```
EOF
```

```
```
```

```
```sql
```

```
sqlite3 provenance.db << 'EOF'
```

```
-- Create the document
```

```
INSERT INTO documents (document_id, document_name)
```

```
VALUES ('doc-001', 'IST Infrastructure KVK');
```

```
-- Create v1.0
```

```
INSERT INTO document_versions (version_id, document_id, version_number, published_by, published_at)
```

```
VALUES ('v1.0-001', 'doc-001', 'v1.0', 'domain-arch-001', datetime('now'));
```

```
-- Link current version
```

```
UPDATE documents SET current_version_id = 'v1.0-001' WHERE document_id = 'doc-001';
```

```
EOF
```

```
```
```

For each ``proposed_amendment`` in your JSON:

```
```sql
```

```
INSERT INTO amendments (
```

```
amendment_id,
```

```
amendment_number,
```

```
document_id,
```

```
submitted_by,
```

```
change_summary,
```

```
change_rationale,
```

```

approval_status,
from_version_id
) VALUES (
'amend-',
'A-001', -- auto-increment this
'doc-001',
'copilot-000',
'[from JSON: change_summary]',
'[from JSON: change_rationale]',
'PROPOSED', -- stays PROPOSED until Domain Architect approves
'v1.0-001'
);

```

For each `implicit\_adr` in your JSON:

```

```sql
INSERT INTO architectural_decisions (
 adr_id,
 adr_number,
 title,
 status,
 context,
 decision,
 consequences,
 proposed_by,
 proposed_at
) VALUES (
 'adr-',
 'ADR-001', -- auto-generate sequentially: ADR-001, ADR-002, ADR-003...
 '[from JSON: adr_title]',
 'PROPOSED', -- stays PROPOSED; Domain Architect decides to ACCEPT or SUPERSEDE
 '[from JSON: where_found]',
 '[from JSON: implications + adr_title combined]',
 '[inferred from document]',
 'copilot-000',
 datetime('now')
);

```

**\*\*Python script to automate this:\*\***

```

```python
import json

```

```

import sqlite3
from datetime import datetime
with open('analysis-2026-07-20.json', 'r') as f:
    data = json.load(f)
    conn = sqlite3.connect('provenance.db')
    cursor = conn.cursor()
    amendment_counter = 1
    for amend in data['proposed_amendments']:
        cursor.execute("""
INSERT INTO amendments (
amendment_id, amendment_number, document_id, submitted_by,
change_summary, change_rationale, approval_status, from_version_id
) VALUES (?, ?, ?, ?, ?, ?, ?, ?)
""", (
f'amend-{amendment_counter:03d}',
f'A-{amendment_counter:03d}',
'doc-001',
'copilot-000',
amend['change_summary'],
amend['change_rationale'],
'PROPOSED',
'v1.0-001'
))
        amendment_counter += 1
        adr_counter = 1
        for adr in data['implicit_adrs']:
            # Get next ADR number sequentially from the database
            # This ensures uniqueness across all ADR runs
            max_adr = cursor.execute(
                "SELECT MAX(adr_number) FROM architectural_decisions"
            ).fetchone()[0]
            next_adr_num = 1 if not max_adr else int(max_adr.split('-')[1]) + 1
            adr_number = f'ADR-{next_adr_num:03d}'
            cursor.execute("""
INSERT INTO architectural_decisions (
adr_id, adr_number, title, status,
context, decision, consequences, proposed_by, proposed_at
) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
""", (
f'adr-{next_adr_num:03d}',

```

```

adr_number,
adr['adr_title'],
'PROPOSED', # Domain Architect decides: ACCEPT or SUPERSEDE
adr['where_found'],
adr['implications'],
f'Criticality: {adr['criticality']}',
'copilot-000',
datetime.now().isoformat()
))
conn.commit()
conn.close()
print(f"Loaded {amendment_counter - 1} amendments and {adr_counter - 1} ADRs")
...
---
```

Step 4: Domain Architect Approves Amendments

Now you review amendments in the database:

```

```sql
-- See all PROPOSED amendments
SELECT amendment_number, change_summary, approval_status
FROM amendments
WHERE approval_status = 'PROPOSED';
...

```

For each amendment:

```

```sql
UPDATE amendments
SET approval_status = 'APPROVED',
approved_by = 'domain-arch-001',
approved_at = datetime('now')
WHERE amendment_number = 'A-001';
...

```sql
UPDATE amendments
SET approval_status = 'REJECTED',
approved_by = 'domain-arch-001',
approved_at = datetime('now')
WHERE amendment_number = 'A-001';
-- (Make changes to the JSON, re-ingest, or create new amendment)
...

```

Leave it as PROPOSED. You'll come back to it.

---

### **Step 5: Generate Next Document Version**

Once amendments are approved:

```
```sql
-- Create v1.1 (incorporates all APPROVED amendments)
INSERT INTO document_versions (
  version_id,
  document_id,
  version_number,
  published_by,
  published_at
) VALUES (
  'v1.1-001',
  'doc-001',
  'v1.1',
  'domain-arch-001',
  datetime('now')
);
-- Link approved amendments to new version
UPDATE amendments
SET to_version_id = 'v1.1-001'
WHERE approval_status = 'APPROVED'
AND document_id = 'doc-001'
AND to_version_id IS NULL;
-- Update current version
UPDATE documents
SET current_version_id = 'v1.1-001'
WHERE document_id = 'doc-001';
```
```

---

### **Workflow Summary**

---

Copilot Analysis

↓

JSON Output

↓

Domain Architect Review

↓ (approved amendments)

Ingest into Database

↓

Generate Document Version v1.1

↓

Output Docx/HTML/Markdown

...

---

## ***Troubleshooting***

**Q:** My JSON doesn't validate

A: Use an online JSON validator ([jsonlint.com](https://jsonlint.com)). Fix syntax errors and re-run Copilot analysis if needed.

**Q:** I want to edit an amendment after ingestion

A: Don't update the amendment directly. Instead, create a *new* amendment that supersedes it:

```
```sql
INSERT INTO amendments (...) VALUES (...);
-- in the new amendment, set a note: "Supersedes A-001"
```
```

**Q:** What if Copilot's analysis is wrong?

A: That's fine. You're the Domain Architect. Edit or reject amendments. The database is your source of truth, not Copilot.

**Q:** Can I undo an approval?

A: Yes. Change `approval\_status` back to `DRAFT` or `PROPOSED`:

```
```sql
UPDATE amendments
SET approval_status = 'DRAFT',
    approved_by = NULL,
    approved_at = NULL
WHERE amendment_number = 'A-001';
```
```

---

## ***Next: Generate Output Formats***

Once you have an approved version in the database, use the output generators to create:

- **Docx** — Word document (for sharing, printing)
- **HTML** — Web page (for wikis, internal sites)
- **Markdown** — Git-friendly format

See *sop-for-klant.pdf* for the complete workflow.

## 5. 5-Phase Implementation

### *Provenance Framework for Domain Architecture*

---

#### **Overview**

This SOP describes how to manage your domain architecture documents using the Provenance Framework. The framework tracks:

- **Amendments** — Changes to the document
- **Architectural Decisions** — The "why" behind decisions
- **Versions** — Clean, published snapshots
- **Releases** — Coordinated feature packages

---

#### **Phase 1: Initial Document Analysis**

- Your existing domain architecture document (PDF or Docx)
- Access to Claude or ChatGPT (Copilot)
- 30 minutes
- 1. Prepare the document**
  - Open your IST Infrastructure document (or SOLL, or any architecture document)
  - Copy the full text or upload the file
- 2. Get the analysis prompt**
  - Open **copilot-analysis-prompt.pdf**
  - Copy the entire prompt
- 3. Run analysis**
  - Go to Claude.ai or your Copilot interface
  - Paste the prompt
  - Attach or paste your document
  - Click "Send" or "Run"
  - Wait for analysis (typically 1-3 minutes)
- 4. Save the JSON output**
  - Copy the JSON result
  - Save as: `analysis-YYYY-MM-DD.json``
  - Validate it (use [jsonlint.com](https://jsonlint.com) if unsure)

You have a JSON file with:

- Document structure (chapters, sections)
- Proposed amendments (one per major section)
- Implicit architectural decisions (ADRs)
- Identified gaps

---

## Phase 2: Domain Architect Review

Your architecture role (e.g., Jaap) reviews the JSON.

**\*\*1. Read the analysis\*\***

- Open the JSON file
- Review each `proposed\_amendment`

**\*\*2. For each amendment, decide:\*\***

| Decision                 | Action                                                                        |
|--------------------------|-------------------------------------------------------------------------------|
| <b>**Accept**</b>        | Leave as-is. Status will be APPROVED in DB.                                   |
| <b>**Needs changes**</b> | Edit the amendment directly in JSON and re-send to Copilot, or edit manually. |
| <b>**Reject**</b>        | Delete the amendment from JSON. Note why in a comment file.                   |

**\*\*3. For implicit ADRs (in the JSON):\*\***

For each `implicit\_adr`:

| Decision                      | Action                                                                           |
|-------------------------------|----------------------------------------------------------------------------------|
| <b>**Accept as decision**</b> | This is a real architectural decision. Status: ACCEPTED (you'll set in Phase 3). |
| <b>**Reject**</b>             | This isn't actually a decision, or it's wrong. Status: SUPERSEDED or delete.     |
| <b>**Formalize later**</b>    | Leave as PROPOSED. You'll review it in Phase 5 and decide to formalize.          |

Examples:

- "We use Kubernetes" → ACCEPT (foundational decision)
- "Security is cross-cutting" → ACCEPT (governance decision)
- "We standardize on X" → ACCEPT (constraint decision)

**\*\*4. For identified gaps (in the JSON):\*\***

- Are these real gaps?
- Should they be tracked as future amendments?

**\*\*5. Approve the JSON\*\***

- Once reviewed, save as: `analysis-YYYY-MM-DD-approved.json`

An approved JSON file ready for database ingestion.

---

## Phase 3: Database Ingestion & Approval

Your architecture role or a designated team member.

- SQLite installed (or accessible)
- `schema.sql` file
- `analysis-YYYY-MM-DD-approved.json` file

**\*\*1. Set up database (first time only)\*\***

```bash

sqlite3 provenance.db < schema.sql

****2. Populate initial data****

```
```bash
```

```
sqlite3 provenance.db << 'EOF'
```

```
INSERT INTO identities (identity_id, name, email, role) VALUES
```

```
('domain-arch-001', '[Architect Name]', '[domain-architect@example.com]', 'Domain Architect'),
```

```
('copilot-000', 'Copilot', 'system@provenance', 'Analysis Tool');
```

```
INSERT INTO documents (document_id, document_name)
```

```
VALUES ('doc-001', 'IST Infrastructure KVK');
```

```
INSERT INTO document_versions (version_id, document_id, version_number, published_by, published_at)
```

```
VALUES ('v1.0-001', 'doc-001', 'v1.0', 'domain-arch-001', datetime('now'));
```

```
UPDATE documents SET current_version_id = 'v1.0-001' WHERE document_id = 'doc-001';
```

```
EOF
```

---

## **\*\*3. Ingest amendments\*\***

Use the Python script in **\*\*amendment-ingestion-guide.pdf\*\***:

```
```bash
```

```
python3 ingest-amendments.py analysis-YYYY-MM-DD-approved.json
```

Or manually insert each amendment (see ingestion guide for SQL).

****4. Review amendments in database****

```
```bash
```

```
sqlite3 provenance.db "SELECT amendment_number, change_summary, approval_status FROM
amendments ORDER BY amendment_number;"
```

---

## **\*\*5. Domain Architect approves amendments\*\***

```
```bash
```

```
sqlite3 provenance.db "UPDATE amendments SET approval_status = 'APPROVED', approved_by =  
'domain-arch-001', approved_at = datetime('now') WHERE amendment_number = 'A-001';"
```

Repeat for all amendments.

****6. Domain Architect reviews and accepts/rejects implicit ADRs****

```
```bash
```

```
sqlite3 provenance.db "UPDATE architectural_decisions SET status = 'ACCEPTED', accepted_by =
'domain-arch-001', accepted_at = datetime('now') WHERE adr_number = 'ADR-001';"
```

```
sqlite3 provenance.db "UPDATE architectural_decisions SET status = 'SUPERSEDED' WHERE
adr_number = 'ADR-001';"
```

---

Review all PROPOSED ADRs before moving to Phase 4.

All amendments in the database with status = APPROVED.

---

## Phase 4: Generate Next Document Version

**\*\*1. Create next version\*\***

```
```bash
sqlite3 provenance.db << 'EOF'
INSERT INTO document_versions (version_id, document_id, version_number, published_by, published_at)
VALUES ('v1.1-001', 'doc-001', 'v1.1', 'domain-arch-001', datetime('now'));
UPDATE amendments
SET to_version_id = 'v1.1-001'
WHERE approval_status = 'APPROVED' AND document_id = 'doc-001' AND to_version_id IS NULL;
UPDATE documents SET current_version_id = 'v1.1-001' WHERE document_id = 'doc-001';
EOF
```
```

**\*\*2. Generate output formats\*\***

Once the next version exists, use the output generators:

```
```bash
python3 generate-docx.py provenance.db doc-001 > IST-Infrastructure-v1.1.docx
python3 generate-html.py provenance.db doc-001 > IST-Infrastructure-v1.1.html
python3 generate-markdown.py provenance.db doc-001 > IST-Infrastructure-v1.1.md
```
```

Three output formats of your document:

- **\*\*IST-Infrastructure-v1.1.docx\*\*** — for sharing, printing
- **\*\*IST-Infrastructure-v1.1.html\*\*** — for wikis, web publishing
- **\*\*IST-Infrastructure-v1.1.md\*\*** — for Git, documentation systems

---

## Phase 5: Creating Next Steps

Domain Architect decides the next phase of evolution.

Once you've published a version (Phase 4), you have:

- ✓ Document version (v1.1, v1.2, etc.)
- ✓ Amendments incorporated
- ✓ ADRs recorded

Now decide:

Some ADRs were "PROPOSED" (generated from the document). Should they become formal ACCEPTED decisions?

**\*\*For each PROPOSED ADR:\*\***

```
```sql
UPDATE architectural_decisions
SET status = 'ACCEPTED', accepted_by = 'domain-arch-001', accepted_at = datetime('now')
WHERE adr_number = '2026-07-001' AND status = 'PROPOSED';
```
```

Or reject it:

```
```sql
UPDATE architectural_decisions
SET status = 'SUPERSEDED'
WHERE adr_number = '2026-07-001' AND status = 'PROPOSED';
```
```

If you want to bundle multiple amendments into a named release:

```
```sql
INSERT INTO releases (release_id, release_number, document_version_id, published_at)
VALUES ('release-002', 'Release-Q3-2026', 'v1.2-001', datetime('now'));
UPDATE architectural_decisions
SET release_id = 'release-002'
WHERE status = 'ACCEPTED' AND adr_number IN ('ADR-001', 'ADR-002', 'ADR-003', ...);
```
```

If the Copilot analysis identified gaps (in `identified\_gaps`), decide:

- **Create new amendments** for each gap
- **Link to ADRs** if the gap requires a decision
- **Schedule for next release**

Example:

```
```sql
INSERT INTO amendments (amendment_id, amendment_number, document_id, submitted_by,
change_summary, approval_status, from_version_id)
VALUES ('amend-gap-001', 'A-100', 'doc-001', 'domain-arch-001', '[gap from analysis]', 'DRAFT',
'v1.1-001');
```
```

Amendments aren't one-time. As your architecture evolves:

1. **Identify a change** (new technology, shifted priorities, etc.)
2. **Create an amendment** (manually or via Copilot)
3. **Domain Architect approves**
4. **Generate next version**

---

---

Published Version (v1.1)

↓

Review PROPOSED ADRs

↓ (formalize or reject)

ACCEPTED ADRs

↓

Plan Next Release (optional)

↓

Identify gaps from analysis

↓

Create amendments for gaps

↓

Repeat: Design → Approve → Publish

...

The database never loses history. Old versions, old amendments, old ADRs — all stay. You only update what's "current" to point to the latest version.

---

## ***Workflow at a Glance***

...

Your Document

↓

Copilot Analysis (Phase 1)

↓

JSON with amendments + implicit ADRs + gaps

↓

Domain Architect Review (Phase 2)

■ Review amendments (approve/reject)

■ Review implicit ADRs (accept/reject)

■ Review gaps (plan next steps)

↓

Ingest into Database (Phase 3)

■ Load amendments as PROPOSED

■ Load ADRs as PROPOSED

■ Domain Architect approves each

↓

Generate Next Version (Phase 4)

↓

Docx + HTML + Markdown outputs

↓

Creating Next Steps (Phase 5)

■ Formalize ADRs (PROPOSED → ACCEPTED)

■ Plan next release

■ Surface gaps as future work

■ Repeat cycle

...

---

## ***Key Files***

| File                            | Purpose                         |
|---------------------------------|---------------------------------|
| `schema.sql`                    | Database schema (run once)      |
| `copilot-analysis-prompt.pdf`   | Template for Copilot analysis   |
| `amendment-ingestion-guide.pdf` | Detailed ingest procedure       |
| `sop-for-klant.pdf`             | This file — end-to-end workflow |

---

## Audit & Security

ClaritasZ (Slot ICT Advies) does NOT see:

- Your documents
- Your analysis JSON
- Your database

You retain full control. ClaritasZ only audits when you ask.

If you want ClaritasZ to audit your provenance:

1. Export a database snapshot
2. Send to ClaritasZ
3. ClaritasZ reviews: amendment discipline, decision tracing, version integrity
4. Feedback returned

---

## Troubleshooting

**Q:** I can't run Python scripts

**A:** Use the SQL approach instead. Copy/paste individual SQL commands.

**Q:** JSON analysis looks wrong

**A:** That's OK. You're the Domain Architect. Edit the JSON or re-run Copilot with corrections. The database is your truth.

**Q:** I want to change an amendment after approval

**A:** Either (a) undo the approval and edit, or (b) create a new amendment that supersedes it.

**Q:** What if I don't have all amendments ready?

**A:** That's fine. Ingest what you have. PROPOSED amendments can wait for review.

**Q:** Can I undo a version?

**A:** The database keeps history. Just don't update `current\_version\_id` to point to an old version. Old versions remain queryable.

**Q:** What's the difference between PROPOSED and ACCEPTED for ADRs?

**A:** PROPOSED = Copilot identified this as a decision embedded in your document. ACCEPTED = You (Domain Architect) verified it's a real decision you want to track formally.

**Q:** Can I link an amendment to an ADR after creation?

**A:** Yes. Update the amendment: `UPDATE amendments SET related\_adr\_id = 'adr-001' WHERE amendment\_number = 'A-001';` (use the internal adr\_id, not adr\_number)

**Q:** What if an amendment generates a new ADR (not detected by Copilot)?

**A:** Create the ADR manually in Phase 5 and link the amendment to it. Both changes are recorded.

---

### Getting Help

- **Copilot analysis questions** → See `copilot-analysis-prompt.pdf`
- **Database ingestion** → See `amendment-ingestion-guide.pdf`
- **SQL questions** → Standard SQLite documentation
- **Framework design** → ClaritasZ (Slot ICT Advies)

---

### Revision History

| on | Date       | Changes                                  |
|----|------------|------------------------------------------|
|    | 2026-07-20 | Initial SOP for initial pilot deployment |

---

**Start with Phase 1. You have everything you need.**



## 6. Terms & Conditions

**\*\*Version 1.0 | July 2026\*\***

---

### 1. Intellectual Property Ownership

The Provenance Framework methodology, including but not limited to the database schema, the amendment discipline, the ADR tracking model, and the audit process, is intellectual property owned exclusively by **\*\*ClaritasZ (Slot ICT Advies)\*\***.

The customer owns and retains exclusive ownership of:

- All documents ingested into the database
- All amendments created and approved by the customer
- All architectural decisions and ADRs specific to the customer's domain
- All outputs generated from customer data (Docx, HTML, Markdown files)

The customer may use, modify, share, and republish these outputs without restriction.

If the customer modifies the Provenance Framework schema, SOP, or code for internal use only, those modifications are owned by the customer and remain confidential. If modifications are to be shared with third parties or published, ClaritasZ must be notified in advance.

---

### 2. Audit Rights

The customer may request an audit of their Provenance database by ClaritasZ at any time. The customer determines:

- **\*\*Frequency\*\*** — How often audits occur (annually, quarterly, ad-hoc)
- **\*\*Scope\*\*** — What aspects are reviewed (amendment discipline, ADR traceability, version integrity, decision provenance)
- **\*\*Timing\*\*** — When the audit takes place

To initiate an audit, the customer provides:

1. A database snapshot (SQLite export) or read-only access to the running database
2. A statement of what aspects the customer wants audited
3. A deadline for audit completion

ClaritasZ reviews the database against the framework principles and deliverables an audit report. The report is confidential to the customer.

ClaritasZ does not monitor, poll, or inspect the customer's database without explicit written request. The customer's database is private infrastructure under their full control.

---

### 3. Commercial Licensing

The Provenance Framework is provided to the customer as a **\*\*pilot implementation\*\*** at no cost. The pilot is intended to validate the framework with the customer's use case and gather feedback for productization.

**\*\*Pilot period:\*\*** July 2026 onwards, until otherwise notified by ClaritasZ.

If and when ClaritasZ decides to commercialize the Provenance Framework as a product:

1. ClaritasZ reserves the right to impose licensing fees for ongoing use of the framework

2. Customers who adopted the framework during the pilot period will be offered favorable pricing and a minimum 12-month transition period before commercial licensing takes effect
3. During the transition period, customers may continue using the framework free of charge
4. A licensing agreement will be negotiated between the customer and ClaritasZ prior to the end of the transition period

A commercial license grants the right to:

- Use the Provenance Framework schema and SOP internally
- Generate customer content (amendments, ADRs, documents)
- Create output artifacts (Docx, HTML, Markdown) for internal or commercial use

A commercial license does NOT grant:

- The right to redistribute the framework code or schema
- The right to sublicense the framework to third parties
- The right to modify and resell the framework as a competing product

**\*\*As of July 2026:\*\*** The framework is provided free of charge. No licensing agreement is in effect. Customers are using the framework as a pilot. ClaritasZ has not committed to any specific commercialization timeline, pricing, or licensing model.

---

#### **4. Data Governance**

The customer owns all data stored in the Provenance database. ClaritasZ has no ownership rights to customer data.

The customer is responsible for:

- Hosting the database (on-premises or cloud infrastructure)
- Maintaining backups
- Securing access to the database
- Ensuring compliance with applicable data protection regulations

ClaritasZ has no access to the database unless explicitly granted by the customer via a snapshot export or read-only access for audit purposes.

The customer determines how long to retain the database and its contents. ClaritasZ places no requirement or limitation on retention.

---

#### **5. Confidentiality**

ClaritasZ treats all customer data, database snapshots, and audit reports as strictly confidential. ClaritasZ will not disclose customer information to third parties without written consent.

The customer agrees not to publicly disclose or reverse-engineer the Provenance Framework methodology, schema, or SOP without ClaritasZ permission.

The customer may:

- Refer to the framework internally within their organization
- Cite the framework in internal documentation and reports
- Describe the framework's benefits and use to prospective customers or stakeholders, provided they do not disclose the internal schema or SOP

---

## **6. Liability and Disclaimers**

The Provenance Framework is provided "as-is" without warranty. ClaritasZ does not warrant that:

- The framework is free of defects
- The framework will meet all customer requirements
- The framework will not conflict with customer IT infrastructure

ClaritasZ's total liability under these terms is limited to the lesser of (a) the fees paid for commercial licensing, or (b) EUR 10,000. This applies to all claims, including negligence, contract, and tort.

Limitations of liability do not apply to:

- Either party's indemnification obligations
- Breaches of confidentiality
- Infringement of intellectual property rights

---

## **7. Term and Termination**

The pilot is indefinite and continues until terminated by either party.

The customer may discontinue use of the framework at any time by ceasing to run the database. No notice is required.

ClaritasZ may terminate the pilot relationship with 30 days' written notice if:

- The customer breaches these terms and does not cure the breach within 30 days
- ClaritasZ decides to commercialize the framework and the customer does not accept the proposed commercial license

Upon termination:

- The customer retains ownership of all customer data
- The customer may continue using the framework under its current deployment (no retroactive licensing)
- Any pilot support from ClaritasZ ceases

---

## **8. Support & Services**

The Provenance Framework is provided as-is without mandatory support. The customer is responsible for installation, configuration, and operation.

ClaritasZ offers optional support on a time-and-materials basis:

- **\*\*Support hours:\*\*** €200/hour (ex. VAT) — billed in 15-minute increments
- **\*\*Audit services:\*\*** €200/hour (ex. VAT)
- **\*\*Scope:\*\*** Setup guidance, troubleshooting, optimization, framework customization

All invoices include 21% Dutch VAT (or applicable VAT rate by jurisdiction).

Support is provided as-needed. ClaritasZ does not guarantee response times or availability.

After the pilot period ends or if commercial use is intended, the customer may either:

- Engage ClaritasZ via hourly support (as above), or
- Negotiate a commercial licensing agreement

---

## **9. Amendments to Terms**

ClaritasZ may amend these terms at any time with 30 days' written notice. Continued use of the framework constitutes acceptance of amended terms.

---

## **10. Governing Law**

These terms are governed by and construed in accordance with Dutch law (jurisdiction: Netherlands).

---

## **11. Contact & Support**

**\*\*ClaritasZ (Slot ICT Advies)\*\***

[Architect Name]

LinkedIn: <https://www.linkedin.com/in/jaapslot/>

Website: <https://claritasz.com>

**\*\*Support & Services\*\***

- Framework: Free download
- Support hours: €200/hour (ex. VAT)
- Contact: LinkedIn or email for quote

---

**\*\*Last Updated:\*\*** July 20, 2026

**\*\*Status:\*\*** Pilot — Free framework + optional support at €200/hour (ex. VAT)



## 7. TOGAF Complementarity

### *Why Architecture Documentation Needs a Provenance Layer*

**\*\*By [Architect Name] | ClaritasZ | July 2026\*\***

---

When we implement TOGAF, we get a methodology. A structured way to think about enterprise architecture. Governance frameworks. Stakeholder analysis. The ADM cycle. Business-to-technology traceability. These are powerful tools.

But when you hand an architect a TOGAF-compliant roadmap and ask them to use it, something gets lost: *\*how did we get here?\**

Not the high-level strategic narrative. That's in the executive summary. But the *\*decisions\** — the hundreds of small, interconnected choices that led from "as-is" to "to-be" — they're implicit. Scattered across meeting notes, emails, documents, and the heads of the people involved. When someone new asks "why do we use Kubernetes instead of traditional VM orchestration?", the answer isn't in the architecture document. It's in the memory of whoever was in the room when that decision was made.

This is the problem that Provenance Framework solves.

---

### *TOGAF Is Normative. Provenance Is Operational.*

TOGAF tells you what the architecture should be. It gives you frameworks for thinking about capabilities, layers, governance chains. It says: "Here are the principles. Here's how you involve stakeholders. Here's how you transition from A to B."

What it doesn't do is operationalize the act of making decisions discoverable.

Provenance Framework doesn't replace TOGAF. It sits underneath it. It's the infrastructure that makes TOGAF decisions visible, traceable, and governable in the long term.

Think of it this way: TOGAF is the strategy. Provenance is the audit trail that says *\*why\** that strategy is correct, and for how long.

---

### *Three Layers of Architectural Thinking*

In a Provenance-managed architecture, three layers emerge:

**\*\*1. Amendments — Observations Made Explicit\*\***

You read the existing architecture. You make amendments. Not judgments. Just clarity: "Here is what the document says." Every time the document changes, that change is an amendment. Who changed it? What changed? Why? All recorded.

This is the closest layer to the raw facts. These are the things we can all agree on without arguing about intent.

**\*\*2. ADRs — Verification of Intent\*\***

Once you've captured what the document says, you ask: "Is this intentional?" Did someone actually *\*decide\** this, or is it an accident? An ADR captures a verified decision. The Domain Architect reviews proposed ADRs and says: "Yes, this is a real decision we made" (ACCEPTED) or "No, this was accidental; let's fix the document" (SUPERSEDED).

This is where intent becomes verifiable.

**\*\*3. Documents — Artifacts of Both\*\***

Finally, the architecture document is generated from the database of amendments and ADRs. Not the other way around. The document is a snapshot of decisions made, not a free-form narrative that hides its reasoning.

This inversion is crucial. In traditional architecture documentation, decisions are buried in prose. You have to re-read the document multiple times to extract them. In a Provenance-managed architecture, the document is *\*derived\** from explicit decisions. It's always consistent with the decisions that created it.

---

### ***How This Complements TOGAF***

**\*\*TOGAF's ADM gives you the method.\*\*** Here's how to approach change: assess the current state, define the target state, plan the transition. It's a structured thinking process. It's normative.

**\*\*Provenance gives you the record.\*\*** Once TOGAF has guided you through the thinking, Provenance captures the result in a way that's:

- **\*\*Traceable:\*\*** Every decision is linked to who made it, when, and why.
- **\*\*Auditable:\*\*** You can ask "has the architecture drifted from the agreed target state?" and have data to answer.
- **\*\*Evolutionary:\*\*** As the organization changes, new decisions supersede old ones. The history remains intact.
- **\*\*Discoverable:\*\*** A new architect joining the team can understand not just the current state, but the reasoning that got us here.

In TOGAF terms, Provenance operationalizes the "governance" part of the framework. It's the system that keeps the architecture honest over time.

---

### ***A Practical Example: Domain Architecture Release Cycles***

Imagine you're a Domain Architect responsible for infrastructure. TOGAF tells you to move from a traditional infrastructure model to a cloud-native model. Good. Now, what does that mean in practice?

With Provenance Framework:

1. **\*\*Analyze the current state.\*\*** An AI-powered analysis extracts what your current architecture document actually says. Every chapter, every decision claim, is captured as an amendment.
2. **\*\*Identify the decisions.\*\*** The Domain Architect reviews those amendments and asks: "Which of these are intentional decisions?" Some are clearly intentional (we *\*chose\** to standardize on Kubernetes). Others are accidental (the document just happens to mention a legacy approach). ADRs capture the intentional ones.
3. **\*\*Plan the transition.\*\*** Based on the verified decisions, the team plans the move to cloud-native. Each change is a new amendment. Each amendment is linked to an ADR (either implementing an existing one or triggering a new decision).
4. **\*\*Release with full traceability.\*\*** The next version of the architecture is released. Every change is auditable. Every decision is linkable. New team members can ask "why did we make this choice?" and get a clear answer.
5. **\*\*Continuous governance.\*\*** Over time, the architecture evolves. Old decisions are superseded. New ones are made. The database keeps the entire history. You never lose track of why something was decided.

This is TOGAF governance made operational.

---

### ***The Missing Piece in Architecture Today***

Most architecture frameworks (TOGAF included) assume that making decisions is the hard part. Once you've decided, you publish the decision in a document and you're done.

But we know that's not how it works. The hard part is \*keeping the decision discoverable years later\*. It's \*understanding why a decision was made when circumstances have changed\*. It's \*onboarding new architects\* without losing the reasoning.

Provenance Framework fills that gap.

It doesn't tell you what the architecture should be (that's TOGAF). It tells you how to make architectural decisions discoverable, auditable, and governable over time.

---

### ***The Three-Layer Model***

If you adopt Provenance alongside TOGAF:

- \*\*TOGAF is your thinking framework.\*\* How to approach architecture systematically.
- \*\*Amendments are your change log.\*\* What changed, by whom, when.
- \*\*ADRs are your decision register.\*\* Which changes represent intentional architectural choices.

Together, they form a complete system: TOGAF gives you the method, Provenance gives you the discipline to keep the decisions verifiable.

---

### ***For Organizations Building Enterprise Architecture***

If you're building architecture practices at scale, you need three things:

1. \*\*A methodology\*\* — TOGAF, DoDAF, or similar. Choose what fits your context.
2. \*\*A decision record\*\* — How you capture that decisions were made. This is where most organizations fail. They rely on documents, emails, and memory.
3. \*\*A governance system\*\* — How you audit and evolve decisions over time.

Provenance Framework is the missing piece — the operational layer that makes #2 and #3 possible.

It's not a replacement for TOGAF. It's a complement. TOGAF tells you \*what\* the architecture should be. Provenance shows you \*how\* to make those decisions discoverable and governable.

---

### ***Starting the Conversation***

If this resonates, ask yourself:

- Can you explain why your current architecture is the way it is?
- Can a new architect understand the reasoning behind major decisions?
- If a decision made five years ago is now questioned, can you trace the rationale?
- Do you know which changes are intentional design choices vs. accidental drift?

If the answer to any of these is "not easily," your architecture is missing a provenance layer.

Provenance Framework provides that layer. It's designed to work alongside TOGAF and other frameworks, not to replace them. It's the practical tooling that makes architectural governance real.

---

\*\*The future of enterprise architecture isn't just better thinking. It's making that thinking verifiable, traceable, and discoverable for years to come.\*\*