FLUX WIKI
DECISIONS LOG

RFC-STYLE RECORD OF CANONICAL PROTOCOL DECISIONS — WHAT WAS DECIDED, WHY, AND WHAT IT CONSTRAINS.

flux.dantesisofo.com/wiki/decisions-log/

FLUX_WIKI_v2.0

JUNE 2026

FLUX DOCUMENTATION SYSTEM Layer 9 — GOVERNANCE | decisions-log
flux.dantesisofo.com/wiki/decisions-log/

DECISIONS LOG

Canonical record of protocol decisions. Each entry is an RFC-style record: problem, decision, rationale, consequences, implementation.

LOCKED decisions cannot be changed without a new DECISION entry that explicitly supersedes and DEPRECATES the prior one. LOCKED is not advisory. It is binding.

DECISION-001: FRAMES_PER_ISSUE = 36

Date: 2026-05-13 Status: LOCKED Category: Protocol

Problem

The FLUX protocol required a fixed frame count per issue. Without a locked count, issues would vary in length, contact sheet grids would vary in geometry, manifest layouts would vary in structure, and PDF page counts would be unpredictable. Consistency is a protocol requirement, not a preference.

Decision

`FRAMES_PER_ISSUE = 36`. This value is fixed. It does not change at runtime. It is not configurable per-photographer or per-issue.

Rationale

36 frames is the capacity of one roll of 35mm film. This is not coincidence. The FLUX protocol derives from film photography's material constraints. 36 frames per roll = 36 frames per issue. The constraint is historical, physical, and meaningful.

A 6×6 contact sheet grid requires exactly 36 cells. The manifest requires exactly 36 entries (2 columns × 18 rows). The PDF structure requires exactly 36 image pages (pages 5–40). These are not coincidences either — the entire protocol is built around this number.

Consequences

- Any system generating FLUX issues must produce exactly 36 frames. No more, no fewer.
- The contact sheet is always 6×6.
- The manifest is always 2 columns × 18 rows.
- The PDF is always 44 pages.
- Auto-generation triggers at exactly 36 unassigned approved photos. Never at 35.
- A FLUX issue with any frame count other than 36 is not a valid FLUX issue.

Implementation

```
# flux_constants.py
FRAMES_PER_ISSUE = 36  # Do not change without updating all dependent scripts
```

Referenced in: `issue_builder_worker.py`, `approve_worker.py`, `generate_flux_issue.py`, `publish_submission.py`, `generate_wiki.py` (contact sheet renderer), PDF manifest generator.

DECISION-002: PDF STRUCTURE — 44 PAGES

Date: 2026-05-13 Status: LOCKED Category: PDF

Problem

The PDF format for a FLUX issue required a fixed page structure. Variable page counts or layouts would make PDFs non-archival — different issues would not be interchangeable, and any system consuming FLUX PDFs would need to handle arbitrary structures.

Decision

Every FLUX PDF is exactly 44 pages. Page assignments are fixed:

```
Page 1:    Front cover
Page 2:    Blank (IFC — inside front cover)
Page 3:    Protocol page
Page 4:    Blank
Pages 5–40: 36 photographs (one per page, sequential)
Page 41:   Parity pad
Page 42:   Contact sheet (6×6 grid)
Page 43:   Manifest (2 columns × 18 rows)
Page 44:   Blank back cover
```

Rationale

Fixed page structure enables: - Predictable print output: any printer handling any FLUX issue gets the same structure - Archival stability: PDF page N always means the same thing in any FLUX issue - Automated processing: any system can parse a FLUX PDF knowing exactly where photographs, manifest, and contact sheet live - Physical production: saddle-stitch binding requires paired pages; 44 pages = 22 spreads = 11 sheets, which folds and staples cleanly

The blank pages (IFC and back cover) are structural requirements for physical binding, not accidents.

Consequences

— PDF generation must always produce exactly 44 pages.

— Page 41 (parity pad) exists to absorb any layout overflow without shifting the contact sheet or manifest.

— Adding or removing page types requires a new DECISION entry and a protocol version bump.

— Any FLUX PDF with a page count other than 44 is malformed.

Implementation

```
# generate_flux_issue.py
PAGES_TOTAL = 44
PAGE_COVER     = 1
PAGE_IFC       = 2
PAGE_PROTOCOL  = 3
PAGE_BLANK_4   = 4
PAGE_IMAGES    = range(5, 41)    # 36 pages
PAGE_PARITY    = 41
PAGE_CONTACT   = 42
PAGE_MANIFEST  = 43
PAGE_BACK      = 44
```

DECISION-003: CANONICAL FILENAME CONVENTION

Date: 2026-05-13 Status: LOCKED Category: Naming

Problem

Photographs entering the FLUX archive from multiple photographers, multiple cameras, and multiple sessions needed a canonical filename format that: - Encoded capture timestamp (for chronological ordering) - Identified the photographer (for attribution) - Preserved the original camera filename (for provenance) - Was human-readable and machine-parseable - Was sortable by filename alone (no metadata lookup required)

Decision

Canonical format: `YYYY-MM-DD_HH-MM-SS_PhotographerName_OriginalFilename.JPG`

Rules: - Date and time are separated by underscore (not ISO 8601 `T`) - Time components use hyphens, not colons (filesystem compatibility) - PhotographerName is CamelCase, no spaces, no hyphens - OriginalFilename is the camera-generated filename, preserved exactly - Extension is always `.JPG` (uppercase) - All four components are separated by single underscores

Examples:

```
2023-01-15_13-22-44_DanteSisofo_R0001234.JPG
2024-07-04_09-15-00_IgorKrivokon_IMG_3421.JPG
2025-11-30_16-45-12_JamesB_DSC09812.JPG
```

Rationale

— Timestamp-first ensures filenames sort chronologically by default

— Underscore-separated components are unambiguous to split by _ with a field count

— Hyphens within date and time fields avoid ambiguity with component separators

— Preserving the original filename preserves provenance — the original camera roll sequence is recoverable

— CamelCase photographer name is human-readable and URL-safe

Consequences

— Filenames are immutable once generated. A canonical filename is a permanent identifier.

— Any system ingesting FLUX photographs must generate this format.

— Filename parsing is deterministic: `split('_', 3)` yields `[date, time, photographer, original]`.

— SHA-256 deduplication supplements filename identity — the same photograph with a different name will be caught by hash.

— Non-compliant filenames are rejected at ingestion.

Implementation

```
# generate_canonical_filename() in approve_worker.py / publish_submission.py
def generate_canonical_filename(captured_at, photographer_name, original_filename):
    date_str = captured_at.strftime("%Y-%m-%d")
    time_str = captured_at.strftime("%H-%M-%S")
    name     = photographer_name.replace(" ", "").replace("-", "")
    return f"{date_str}_{time_str}_{name}_{original_filename}"
```

DECISION-004: S3 NAMESPACE SEPARATION

Date: 2026-05-13 Status: ACTIVE Category: Infrastructure

Problem

The FLUX S3 bucket (`flux-dantesisofo`) needed to host multiple asset types — full-resolution photographs, thumbnails, generated PDFs, and catalog assets — without namespace collisions.

Decision

S3 prefix assignments:

```
photos/          — full-resolution personal photographs
thumbs/          — generated thumbnails (personal archive)
FLUX_ISSUES/     — personal archive PDF issues (FLUX_NNN/)
FLUX_CATALOG/    — public catalog assets (CAT_NNN/)
```

Rationale

Prefix separation enables: - Independent IAM policies per prefix - Independent CloudFront cache behavior configurations per prefix - Clean `ListObjectsV2` calls scoped to a single asset type - Safe bulk operations (delete all thumbs, sync all issues) without risk of cross-contamination

Consequences

— `FLUX_ISSUES/FLUX_001/`, `FLUX_ISSUES/FLUX_002/`, etc. — personal issues are always under `FLUX_ISSUES/`

— FLUX_CATALOG/CAT_001/, FLUX_CATALOG/CAT_002/, etc. — catalog entries are always under FLUX_CATALOG/
— New asset types require a new prefix decision (not a free-form naming choice)
— Mixing asset types under the same prefix is a protocol violation

Implementation

```
# deploy_s3.py / publish_submission.py
PERSONAL_PREFIX  = "photos/"
THUMBS_PREFIX    = "thumbs/"
ISSUES_PREFIX    = "FLUX_ISSUES/"
CATALOG_PREFIX   = "FLUX_CATALOG/"
```

DECISION-005: CAT_NNN VS FLUX_NNN NAMESPACE SEPARATION

Date: 2026-05-14 Status: LOCKED Category: Naming

Problem

The FLUX protocol serves two distinct use cases: 1. Dante Sisofo's personal photographic archive (continuous, private, 423+ issues) 2. The public FLUX catalog — issues submitted by any photographer

These needed separate identifier namespaces. Using a single sequence (e.g., FLUX_001 for both) would create ambiguity about whether an identifier referred to a personal or public issue.

Decision

Personal archive identifiers: FLUX_NNN (FLUX_001, FLUX_002, ..., FLUX_423, ...) Public catalog identifiers: CAT_NNN (CAT_001, CAT_002, ..., CAT_018, ...)

The namespaces are completely separate. A CAT_NNN entry is not a FLUX_NNN issue and never becomes one.

Rationale

The personal archive is a private, continuous, chronological record. The public catalog is a curated collection of external submissions. These are different things. Conflating their identifiers would falsify the record. A submission from another photographer is not Dante Sisofo's personal archive.

Consequences

— _next_catalog_id() generates CAT_NNN identifiers only
— _next_issue_id() generates FLUX_NNN identifiers only
— Neither function can generate the other's namespace
— Public catalog count: 18 (CAT_001–CAT_018) as of 2026-05-20
— Personal archive count: 423+ (FLUX_001–FLUX_423+) as of 2026-05-20

Implementation

```
# publish_submission.py
CATALOG_ID_PREFIX = "CAT_"

# issue_builder_worker.py
ISSUE_ID_PREFIX = "FLUX_"
```

DECISION-006: CATALOG.JSON AS CATALOG SOURCE OF TRUTH

Date: 2026-05-14 Status: ACTIVE Category: Infrastructure

Problem

The public catalog required a persistent, authoritative record of all catalog entries. This record needed to be machine-readable, human-readable, and deployable to S3 for the live website.

Decision

`catalog.json` is the single source of truth for all public catalog entries. Every catalog read and write goes through this file. No catalog operation is complete until `catalog.json` is updated and synced to S3.

Rationale

A single JSON file is simple, version-controllable, and directly readable by the static site. No database dependency for the public catalog. No API dependency for catalog reads.

Consequences

- `catalog.json` must be synced to S3 on every write (see DECISION-008)
- `catalog.json` must be preferred over local disk when the two diverge (see DECISION-008)
- The `_next_catalog_id()` function must NOT derive IDs from `catalog.json` (see DECISION-007)
- Any catalog backup/recovery operation targets `catalog.json` plus its S3 object versions

Implementation

```
# publish_submission.py
CATALOG_JSON_LOCAL = "/path/to/catalog.json"
CATALOG_JSON_S3    = "catalog.json"  # root of bucket, served by CloudFront
```

DECISION-007: _next_catalog_id() MUST SCAN S3 FOLDERS

Date: 2026-05-19 Status: LOCKED Category: Infrastructure

Problem

`_next_catalog_id()` was computing the next catalog identifier by reading `max(catalog.json entries) + 1`. This caused a critical failure: if `catalog.json` was not current with S3 (due to a deferred or failed sync), the function would generate an ID that was already in use, overwriting existing catalog entries.

CAT_002 and CAT_003 were overwritten by this bug. Both were recovered from S3 object versioning.

Decision

`_next_catalog_id()` must scan S3 `FLUX_CATALOG/` folder prefixes directly using `ListObjectsV2` with `Delimiter='/'`. It must derive the next ID from `max(existing_CAT_NNN_prefixes) + 1`. It must not consult `catalog.json` for ID generation.

Rationale

S3 is the authoritative record of what catalog entries exist. `catalog.json` is a derived view that can be stale. The ID space must be derived from the authoritative source, not from a potentially stale cache.

Consequences

— `_next_catalog_id()` requires a live S3 API call on every invocation. This is intentional.

— Network failure during ID generation is a hard error — the operation cannot proceed without a live S3 scan.

— The function is slower than reading from a local file. This is acceptable. Correctness is not negotiable.

— CAT_NNN IDs are guaranteed unique as long as S3 is the authoritative namespace.

Implementation

```
def _next_catalog_id(s3_client, bucket):
    paginator = s3_client.get_paginator('list_objects_v2')
    pages = paginator.paginate(Bucket=bucket, Prefix='FLUX_CATALOG/', Delimiter='/')
    existing = []
    for page in pages:
        for prefix in page.get('CommonPrefixes', []):
            folder = prefix['Prefix']  # e.g., 'FLUX_CATALOG/CAT_018/'
            match  = re.search(r'CAT_(\d+)/', folder)
            if match:
                existing.append(int(match.group(1)))
    next_n = (max(existing) + 1) if existing else 1
    return f"CAT_{next_n:03d}"
```

DECISION-008: _save_catalog() MUST IMMEDIATELY SYNC TO S3

Date: 2026-05-19 Status: LOCKED Category: Infrastructure

Problem

`_save_catalog()` was writing `catalog.json` to local disk and deferring the S3 upload. In some code paths, the S3 upload was skipped entirely. This caused local and S3 versions to diverge. `_load_catalog()` was preferring local disk over S3. The combination created a state where the local system had stale catalog data and generated duplicate CAT IDs.

Decision

`_save_catalog()` must upload `catalog.json` to S3 immediately on every write, before returning. The function is not complete until S3 is updated.

`_load_catalog()` must prefer the S3 version over local disk. On load, both versions are fetched; whichever has more entries is used; the local disk is updated to match.

Rationale

S3 must be the authoritative source. Local disk is a cache. Caches are not authoritative. The system must treat them as such at all times, not only when convenient.

Consequences

— Every catalog save requires a network round-trip to S3. This is a hard requirement.

— S3 write failures on `_save_catalog()` are hard errors. The catalog state is undefined if S3 is not updated.

— `_load_catalog()` on a system with no local disk copy still works correctly (loads from S3).

— The system is safe to restart at any point — the authoritative state is always on S3.

Implementation

```
def _save_catalog(catalog, s3_client, bucket, local_path, s3_key):
    with open(local_path, 'w') as f:
        json.dump(catalog, f, indent=2)
    s3_client.put_object(
        Bucket=bucket,
        Key=s3_key,
        Body=json.dumps(catalog, indent=2).encode('utf-8'),
        ContentType='application/json'
    )
    # S3 upload must complete before function returns

def _load_catalog(s3_client, bucket, local_path, s3_key):
    local  = _load_local(local_path)
    remote = _load_s3(s3_client, bucket, s3_key)
    canonical = remote if len(remote) >= len(local) else local
    _write_local(local_path, canonical)
    return canonical
```

DECISION-009: AUTO-GENERATION TRIGGER AT FRAMES_PER_ISSUE THRESHOLD

Date: 2026-05-15 Status: ACTIVE Category: Infrastructure

Problem

The FLUX portal required an automated pathway from approved photograph accumulation to issue generation. Manual triggering would create bottlenecks and require constant monitoring.

Decision

When the count of unassigned approved photographs in the queue reaches `FRAMES_PER_ISSUE` (exactly 36), `approve_worker.py` spawns `issue_builder_worker.py` as a detached subprocess. A file lock at `/tmp/flux_issue_builder.lock` prevents duplicate builds.

Rationale

The threshold is exact: 36, not "more than 36." The system triggers precisely when one full issue worth of photographs is available. This is the simplest correct implementation of the protocol.

File locking prevents race conditions when multiple approve operations complete in rapid succession.

Consequences

— The trigger fires at exactly 36 unassigned approved photos. It does not fire at 35.

— If the lock file exists when the trigger fires, the new build is skipped (the previous build will consume the photos).

— The lock file is created at build start and removed at build completion (normal or abnormal exit).

— Stale lock files (from crashed builds) must be detected and cleaned up. This is a known operational risk.

— The auto-generation pathway is: portal approval → approve_worker.py → _maybe_trigger_build() → issue_builder_worker.py.

Implementation

```
# approve_worker.py
LOCK_FILE = "/tmp/flux_issue_builder.lock"

def _maybe_trigger_build(unassigned_count):
    if unassigned_count < FRAMES_PER_ISSUE:
        return
    if os.path.exists(LOCK_FILE):
        return  # build already in progress
    subprocess.Popen(
        ["python3", "/path/to/issue_builder_worker.py"],
        start_new_session=True,
        stdout=subprocess.DEVNULL,
        stderr=subprocess.DEVNULL,
```

)

DECISION-010: NAS = CANONICAL ARCHIVE ROOT, MAC MINI = COMPUTE, S3 = DISTRIBUTION

Date: 2026-05-20 Status: ACTIVE Category: Infrastructure

Problem

The FLUX archive requires three distinct capabilities: durable primary storage, compute for PDF generation and image processing, and public distribution via CDN. These are different requirements with different failure modes and cost profiles. Running them on a single machine creates single points of failure and performance conflicts.

Decision

Three-node architecture: - **NAS (Synology, 2x mirrored IronWolf):** canonical archive root. All photographs, all issues, all metadata, all embeddings live here. This is the source of truth for the physical archive. - **Mac mini M4 Pro:** compute node. Runs all processing — ingest pipeline, PDF generation, embedding generation, keeper model training, portal server. Stores no canonical data; all outputs are written to NAS or S3. - **AWS S3 + CloudFront:** public distribution. Serves the live website, PDFs, and photographs to the public. NAS is the source; S3 is the published mirror.

Rationale

NAS mirroring protects against single-drive failure. Mac mini M4 Pro has sufficient compute for the full FLUX pipeline (PDF generation, embedding inference, model training). S3 + CloudFront provides CDN-grade public delivery without requiring the NAS to be internet-accessible.

This architecture has no single point of failure for the archive: if Mac mini fails, the archive on the NAS is intact. If S3 goes down, the archive on the NAS is intact. If the NAS loses one drive, the mirror continues.

Consequences

— The NAS folder structure is the canonical organization of the archive (see CHANGELOG v2.0).

— The Mac mini mounts the NAS over the local network. All scripts use NAS-mounted paths.

— No canonical data is stored only on the Mac mini's local SSD.

— S3 is a publication layer, not a backup. The NAS is the backup.

— Disaster recovery: restore the NAS from its own mirror, re-sync to S3, restart Mac mini processes.

Implementation

NAS folder structure:

```
/FLUX_ARCHIVE/      — full corpus + keepers (~400,000 frames)
/FLUX_SYSTEM/       — scripts, configuration, virtualenvs
/FLUX_PUBLIC/       — S3 sync source (generated site assets)
/FLUX_METADATA/     — SQLite database, manifests
/FLUX_EMBEDDINGS/   — vector database files
/FLUX_ISSUES/       — generated PDF issues
/FLUX_LOGS/         — all process logs
/FLUX_INBOX/        — incoming photos for ingest
```

DECISION-011: HUMAN STAPLING AS FINAL PHYSICAL STEP

Date: 2026-05-15 Status: ACTIVE Category: Protocol

Problem

Physical FLUX issue production requires saddle-stitch binding. The question was whether stapling should be automated (via a connected electric stapler triggered by the print pipeline) or performed manually by the photographer.

Decision

Stapling is a human step. The print pipeline delivers the folded sheets. The photographer staples.

Rationale

The physical act of assembling the issue is part of the protocol. FLUX issues are not produced by a machine end-to-end. The photographer shoots, selects (currently manual, eventually model-assisted), approves (portal), and assembles (stapling). The assembly step belongs to the photographer.

Automated stapling would require a network-connected stapler, additional failure modes, and additional calibration. The marginal time savings do not justify the complexity.

Consequences

- The physical print pipeline ends at: PDF generated → sent to printer → pages emerge → photographer folds → photographer staples.
- No automation is planned for the stapling step.
- This decision is revisable if production volume requires it. At 423+ issues per photographer, the time cost of manual stapling is known and acceptable.

Implementation

Operational, not code. The print pipeline sends the PDF to the network printer via system print command. Physical assembly is the photographer's responsibility.

SEE ALSO

Document	Layer	Relationship
CHANGELOG	Layer 9 — Governance	Chronological version history for all decisions above
VERSIONS	Layer 9 — Governance	Version numbering and locked-version semantics
PROTOCOL	Layer 2 — Protocol	The canonical method that DECISION-001 and DECISION-002 define
BOOTSTRAP	Layer 4 — Infrastructure	Phased implementation plan for DECISION-010 hardware
ARCHIVE	Layer 2 — Protocol	Digital archive structure that DECISION-003 and DECISION-004 govern

FLUX_WIKI_v2.0 — flux.dantesisofo.com/wiki/decisions-log/