Photo Editor
Low-Level Design & Detailed Test Plan

Krishhiv Mehra & Siddhant Rau

10 November 2025

Document Purpose. This specification expands the previously submitted High-Level Design (Assignment 4) by detailing module internals (functions and classes), on-wire contracts, and a comprehensive test plan. Any architectural adjustments since, are folded directly into the relevant sections below.

1 Detailed Component & Module Design

1.1 Frontend - NextJS

- **Framework & State.** NextJS (App Router) with React Query for server state[1] and Zustand for editor UI state.
- **Key Routes.** `/library`, `/albums/[id]`, `/editor/[photoId]`, `/share/[token]` (public share with SSR[2]).
- **Core Components.** *Uploader*, *ImageCanvas* (Fabric.js/Konva adapter), *ToolSidebar*, *HistogramPanel*, *OpGraphInspector*, *ShareDialog*.
- **Editor State.** A pure JSON *operation graph* ("op-graph") represents the sequence of edits. The canvas applies the same transforms client-side for instant preview; no pixels are persisted until *commit*.
- **Safe Retrieval.** Originals/variants are downloaded via signed Azure Blob SAS[3] URLs. Public share pages are SSR with cache headers.
- **Validation.** Zod schemas mirror backend Pydantic models to prevent drift.

1.2 API Service (FastAPI)

Responsibilities. Auth, uploads, albums/photos, edit commit, search, sharing, and metadata (EXIF[4]/ICC[5]) handling.

Package Layout:

- `api/main.py` — app factory, middleware (CORS/HSTS), request ID logging.
- `api/deps.py` — auth dependencies, DB session management.

[1]Server State: data fetched from remote APIs and cached on the client for synchronization.
[2]SSR: Server-Side Rendering.
[3]SAS: Shared Access Signature.
[4]EXIF: Exchangeable Image File Format.
[5]ICC: International Color Consortium (color profiles).

- `api/config.py` — typed settings via `pydantic-settings`.
- `api/routers/auth.py` — login/refresh (JWT[6]).
- `api/routers/uploads.py` — SAS issuance; chunked block-blob completion.
- `api/routers/albums.py` — CRUD + ACL[7] checks.
- `api/routers/photos.py` — ingest, metadata read, listing, search.
- `api/routers/edits.py` — op-graph validation, commit, versioning.
- `api/routers/search.py` — tag/EXIF queries, pagination.
- `api/routers/shares.py` — share token issue/revoke, scope & expiry.
- `api/schemas/*.py` — Pydantic DTOs (`album.py`, `photo.py`, `version.py`, `share.py`, `errors.py`).
- `api/services/image_service.py` — Pillow/OpenCV transforms, tiled processing.
- `api/services/storage_service.py` — Azure Blob put/get via SAS.
- `api/services/version_service.py` — variant key derivation & metadata persistence.
- `api/services/exif_service.py` — EXIF read/strip GPS; ICC preservation.
- `api/repos/*.py` — SQLAlchemy ORM[8] repositories for users, albums, photos, versions.
- `api/utils/opgraph.py` — schema, canonicalization, hashing.
- `api/utils/security.py` — JWT, share tokens (hashed at rest).
- `api/utils/idempotency.py` — Idempotency-Key enforcement for safe retries.

1.3 Database (PostgreSQL) – Schemas & Columns

We use PostgreSQL with SQLAlchemy 2.0. Naming: *snake_case* tables; `created_at/updated_at` timestamps; FKs with `ON DELETE CASCADE`. A partial GIN[9] index supports tag search.

Schema: `core`

Table	Column (type)	Why it exists
users	id (uuid, PK)	Primary identity for auth & ownership.
	email (text, unique)	Unique login; lookup key.
	password_hash (text)	Store Argon2/BCrypt hash, not plaintext.
	created_at (timestamptz)	Auditing and sort by recency.
albums	id (uuid, PK)	Logical grouping of photos for users/teams.
	owner_id (uuid, FK users.id)	Ownership & ACL checks.
	title (text)	Human-readable label.

[6]JWT: JSON Web Token.
[7]ACL: Access Control List.
[8]ORM: Object-Relational Mapping.
[9]GIN: Generalized Inverted Index.

Table	Column (type)	Why it exists
	created_at (timestamptz)	Ordering & lifecycle ops.
photos	id (uuid, PK)	Stable identity for an uploaded original.
	album_id (uuid, FK albums.id)	Belongs-to relation.
	blob_path (text)	Pointer to Blob URL path.
	mime_type (text)	Validate processing/serving.
	width, height (int)	Prevent illegal ops; UI layout.
	exif (jsonb)	Persist useful camera metadata.
	created_at (timestamptz)	Traceability.
versions	id (uuid, PK)	Each committed edit = a version.
	photo_id (uuid, FK photos.id)	Back-reference to original.
	op_graph (jsonb)	Audit the exact edits applied.
	variant_key (text, unique)	Address immutable CDN[10] path.
	format (text)	Serving/analytics; e.g., jpg/png/webp.
	bytes (int)	Size tracking for quotas.
	created_at (timestamptz)	Ordering/history.
shares	id (uuid, PK)	Public/limited access token record.
	photo_id (uuid, FK photos.id)	What is being shared.
	scope (text)	`view` or `comment` etc.
	token_hash (text)	Store a hash, not raw token.
	expires_at (timestamptz)	Auto-expiry enforcement.

Schema: `tags`

Table	Column (type)	Why it exists
photo_tags	photo_id (uuid, FK photos.id)	Many-to-many mapping from photo to tag.
	tag (text)	Search & organization; GIN indexed.

Indices & Constraints.

- `users.email` unique; `versions.variant_key` unique (*idempotency cache*).
- GIN index on `photos.exif` keys commonly queried; partial index on `tags.photo_tags(tag)`.
- `ON DELETE CASCADE` for `albums`→`photos` and `photos`→`versions`.

[10]CDN: Content Delivery Network.

2 Operation Graph - Clear Explanation

2.1 What it is

An **operation graph** is a compact JSON description of *what* edits to apply and in *which order.* Think of it as a recipe; the original image plus this recipe renders the same output every time. This enables:

- **Reproducibility.** Same inputs $\Rightarrow$ same result.
- **Caching.** We compute a short hash of the recipe to create a reusable *variant key.*
- **Auditability.** You can always see which edits produced a version.

2.2 Minimal Example

```
{
  "photoId": "uuid-1234",
  "ops": [
    {"op":"crop","x":120,"y":90,"w":1024,"h":768},
    {"op":"rotate","deg":90},
    {"op":"exposure","ev":0.33}
  ],
  "output": {"maxW": 2048, "format": "auto", "quality": "auto"}
}
```

The engine applies `crop` then `rotate` then `exposure`, and finally chooses an output encoding (see Section 3).

2.3 Canonicalisation

To make caching reliable we canonicalise the JSON before hashing:

1) Round numeric fields to three decimals; normalize colors to #rrggbb.
2) Freeze the operation order where it matters (e.g., exposure before contrast).
3) Remove irrelevant whitespace and sort object keys.

Result: identical edits *always* produce the same canonical JSON and therefore the same hash & filename.

2.4 Hash and Variant Key

We compute `hash = BLAKE3(photoId || canonical_json)` and set `variant_key = v/{photoId}/{hash[0..9]}.{ext}`. This makes every version addressable at an immutable path.

2.5 Supported Operations

- `crop`: `x,y` ≥ 0; `w,h` ≥ 1.
- `rotate`: degrees in $[-180, +180]$; bicubic resampling.
- `scale`: `maxW/maxH` > 0; Lanczos downscale; upscale capped at $2\times$.
- Tone/Color: `exposure`, `brightness`, `contrast`, `saturation`, `hue` within bounded amounts in linear RGB with ICC preserved.

- Blur/Sharpen: gaussian $\sigma \in [0, 5]$; unsharp mask (radius, percent, threshold).
- Text/Watermark: font allowlist; color contrast defaults meet AA[11].

3 Image Processing & Adaptive Encoding

- **Libraries.** Pillow/OpenCV; optional AVIF plugin. Tiled processing avoids excessive memory for very large images.
- **Color.** Adjust tone/contrast in linear RGB; convert back to sRGB; keep ICC; strip GPS from EXIF on public variants.
- **Formats.** JPEG (progressive, 4:2:0) for photos; WebP for size gains $\geq$15%; PNG or WebP-lossless for graphics/alpha.
- **Heuristic.** Probe on a 256px sample to select target format; encode full-res once.
- **Quality Gates.** Golden PSNR[12] $\geq$ 38 dB or SSIM[13] $\geq$ 0.98 for auto-lossy; else fallback to lossless.

4 Custom Protocols & Contracts

- **Upload.** Client requests SAS; uploads via Azure Block Blob (4–8 MB blocks in parallel). Client posts a manifest to `/uploads/complete`; server verifies and persists metadata.
- **Edit Commit (Idempotent).** `POST /photos/{id}/edits` with op-graph JSON and `Idempotency-Key` header; server returns a new *version* with a signed URL.
- **Variant Retrieval.** Immutable GET to `v/{photoId}/{hash}.{ext}` with long-lived cache headers via CDN.

5 Low-Level Sequences

5.1 Commit Edit

1) Validate auth, ACL, and `photoId`; load photo metadata.
2) Validate & canonicalize op-graph; compute hash & variant key.
3) If variant already exists, persist only the `versions` row (idempotent fast-path).
4) Render pixels (tiled); choose encoding; upload variant to Blob.
5) Persist `versions` row and return DTO with signed URL.

5.2 Upload

1) Client obtains SAS for path `originals/{userId}/YYYY/MM/uuid.ext`.
2) Client uploads blocks with retries/backoff; completes with manifest.
3) API verifies, sniffs type/dimensions, writes `photos` row, pre-generates thumbnails.

[11]WCAG AA: Web Content Accessibility Guidelines, Level AA.
[12]PSNR: Peak Signal-to-Noise Ratio.
[13]SSIM: Structural Similarity Index Measure.

6 Test Strategy & Quality Gates

6.1 Methodology

Test Pyramid: heavy unit & property tests; targeted integration; concise E2E[14] via Playwright; separate performance and security suites. Environments: local (SQLite + Azurite), CI (PostgreSQL + Azurite), staging (Azure).

6.2 Unit Tests (PyTest + Hypothesis)

- **opgraph.** Canonicalisation (ordering, rounding), hashing determinism, invalid ranges rejected.
- **image_service.** Each op validated on synthetic images; rotation invariants; bounds.
- **version_service.** Stable key derivation; collision checks.
- **storage_service.** SAS scoping/expiry; MIME sniffing; type mismatch handling.
- **repos.** CRUD with constraint violations & rollbacks.

6.3 Integration Tests

- **Upload.** Chunked upload via Azurite; `photos` row exists; thumbnails reachable.
- **Edit Commit.** Given an original, post op-graph; confirm variant exists; bytes stable; ICC present; EXIF GPS stripped for public.
- **Search.** Tag + EXIF filters return expected sets with stable pagination.
- **Auth/ACL.** Share token scope respected; expiry enforced.

6.4 E2E

- Scenario: signup → upload → crop/rotate → save → open share link (mobile viewport).
- Accessibility: axe-core checks; keyboard navigation; dialog focus traps.

6.5 Performance & Reliability

- **Load.** Locust: 10→100 users editing distinct 24MP photos; record p50/p95; error rates.
- **Soak.** 1-hour continuous uploads of 10MB images with 1% network failure injection; verify no orphaned rows/objects.
- **Memory Ceiling.** Per-process RSS $<$ 60% of VM during tiled operations.

6.6 Security Testing

- JWT expiry/forgery; refresh rotation; nonce checks.
- SAS abuse prevention (path scoping, minimal permissions); optional IP policy.
- OWASP vectors: malformed/oversized uploads; content-type confusion; SVG/script injection; CSP[15] verified.
- Privacy: EXIF GPS stripping; no sensitive headers in responses.

[14]E2E: End-to-End.
[15]CSP: Content Security Policy.

7 Representative Test Cases

- **TC-U-001** Op-graph canonicalization ⇒ expect rounded JSON and stable hash.
- **TC-U-012** Rotate 90° on 8×8 checkerboard ⇒ pixel-perfect result.
- **TC-I-021** 50MB JPEG chunked upload (reordered blocks) ⇒ server reassembles; DB row created; thumbnails present.
- **TC-I-034** Edit commit idempotency with same `Idempotency-Key` ⇒ single version row; identical payload.
- **TC-E-005** Share link with past `expires_at` ⇒ 403 forbidden.
- **TC-P-009** p95 edit latency for 24MP crop+rotate+exposure $\leq$ 2.5s on B2s VM[16].
- **TC-S-007** Malicious SVG upload ⇒ rejected with 415/422; sanitized preview only.

8 Requirement-to-Test Traceability

```
R-UP-01 (Chunked uploads) -> TC-I-021, E2E Scenario
R-ED-02 (Deterministic edits) -> TC-U-001, TC-I-034, golden-image suite
R-SH-03 (Share scopes & expiry) -> TC-E-005, E2E Scenario
R-PR-04 (Privacy: GPS stripping) -> Integration + Security tests
R-PF-05 (p95 <= 2.5s) -> TC-P-009 (Locust run)
```

9 Operational Concerns

- **Observability.** Structured JSON logs (reqId, route, status, latency); optional Sentry; Azure Monitor counters.
- **Backups.** Nightly `pg_dump`; Blob lifecycle policy; quarterly restore drills.
- **Reconciliation.** Orphan sweeper for DB ↔ Blob mismatches with dry-run reports.

ChatGPT 5 was used to research and for reference purposes.

[16] VM: Virtual Machine.