

Building FrameShift

From One Mac to a Render Operations Platform

A practical engineering account of local-first architecture, durable scheduling, secure Blender automation, and product design

Book Edition

First Edition · September 2026

Built on macOS with Go, Next.js, Blender, FFmpeg, SQLite, PostgreSQL, and a great many carefully tested boundaries.

Publication Note

This book documents the design and implementation of FrameShift as it existed during its first substantial development cycle. It is an engineering narrative, not merely a feature catalogue and not a substitute for the repository itself. Source code, dependencies, operating systems, native tools, and deployment practices continue to evolve. Where a precise runtime fact matters, the current code and its tests remain authoritative.

The examples in these pages are deliberately selective. They illustrate boundaries, decisions, and failure modes without reproducing the whole application. Paths, identifiers, configuration values, and compact excerpts are shown to make the discussion concrete. Local development credentials are examples only and must be replaced anywhere a FrameShift installation crosses a trusted personal machine boundary.

Blender is a trademark of the Blender Foundation. macOS and Apple Silicon are trademarks of Apple Inc. PostgreSQL, FFmpeg, Docker, Next.js, Go, and the other tools discussed in this book belong to their respective projects and owners. Their inclusion describes the FrameShift implementation and does not imply endorsement.

Author: *[Add author name before publication]*

Edition: First edition, September 2026

Repository: FrameShift project source and documentation

Dedication

For artists who have watched a long render sit at zero percent and wondered whether anything was happening, and for engineers who believe local tools deserve the same care as distributed systems.

If a system cannot explain and recover from work on one computer, adding ten computers will only distribute the confusion.

Preface

FrameShift began with a bounded but revealing question: **how can one Mac run Blender work with the visibility, recovery, and control expected from a managed service, without first requiring a cloud control plane?**

The answer became much larger than a process launcher. A reliable render platform has to reason about untrusted creative files, expensive native processes, partial output, durable state, user identity, worker ownership, retries, media finalization, storage retention, collaboration, and recovery. Every boundary can fail independently. A browser can refresh while Blender continues. A worker can disappear while a lease remains. A frame can be valid while the final video is not. An email can fail after a job has completed. A backup can contain a database but omit the files that make its rows meaningful.

This book follows the complete implementation from that first constraint to a working render operations platform. It asks not only *what was built*, but *why it was built this way*. Why did Go become the authoritative backend rather than Node.js, Python, or Rust? Why did SQLite make sense before PostgreSQL? Why does a worker lease a frame instead of simply taking ownership of a job? Why are uploaded Blender files treated as hostile input? Why is FFmpeg finalization a durable stage rather than an incidental command at the end of a worker function? Why do logs, retries, approvals, and storage tiers belong to the product experience instead of an operator-only afterthought?

The central rule never changes: **the Go backend and its durable store decide authoritative state**. The Next.js application sends intent and presents persisted facts. Workers claim leased tasks, supervise Blender, and report evidence. Blender, FFmpeg, FFprobe, Mailpit, PostgreSQL, and MinIO retain specialist responsibilities behind explicit interfaces without becoming alternate schedulers or informal sources of truth.

That rule is simple to state, but the chapters ahead show how much discipline it demands. The frontend cannot declare a job complete because a progress animation reached one hundred percent. A worker cannot declare a job complete because Blender exited successfully. A file appearing in an output directory cannot declare itself authoritative. Each participant knows only part of the story. The backend combines those observations inside transactions, validates ownership and state transitions, and records the result in a form that survives process restarts.

Why This Book Exists

Many software accounts begin after the architecture has become neat. They present the final component diagram, list the technologies, and leave out the uncertain middle where native tools behave differently than expected, migration assumptions fail, browser state becomes stale, or a

seemingly harmless file format turns out to have several valid containers. This book keeps that middle visible.

You will encounter decisions that were revised after real evidence. Blender 5.1 changed assumptions about render-engine identifiers and compressed `.blend` files. A static zero-percent display proved inadequate for a single long frame, leading to streamed logs and sub-frame sample progress. A worker restart exposed the difference between a parent `go run` process and its compiled child. A PostgreSQL smoke test found SQL and timestamp behavior that SQLite tests could not reveal. An accessibility audit turned visual preferences into measurable contrast and semantic fixes.

These are not embarrassing footnotes. They are the substance of engineering. The value of the final design comes partly from the failures that made hidden assumptions explicit.

What the Reader Will Learn

The first parts establish the product problem, local-first constraint, technology choices, and authority model. The middle of the book follows a scene through secure ingestion, immutable versioning, durable scheduling, worker execution, real-time observability, frame validation, FFmpeg finalization, artifact management, and publishing. The later parts examine identity, teams, email, review approval, backups, PostgreSQL, object storage, deployment, user-interface design, accessibility, and testing.

Although FrameShift is a Blender render manager, its engineering patterns apply more broadly. Any system that coordinates expensive subprocesses can benefit from leased work, durable attempts, atomic promotion, explicit state machines, bounded logs, idempotent delivery, and honest recovery states. Media pipelines, local AI tools, scientific workloads, build systems, and desktop automation platforms face many of the same questions.

How to Read This Book

The ten parts form one continuous path and are best read in order. Parts 1 through 3 explain the premise and architecture. Parts 4 through 7 follow the operational journey from uploaded bytes to published deliverables. Parts 8 through 10 cover collaboration, operation, interface design, testing, and the lessons that emerged from building the system.

Readers may also enter at a particular boundary. Security-focused readers can begin with Part 4. Engineers interested in queues and failure recovery can begin with Part 5. Blender and macOS integration is concentrated in Part 6. Media delivery and FFmpeg appear in Part 7. Identity and collaboration are covered in Part 8, deployment in Part 9, and product/testing concerns in Part 10.

Code blocks are representative excerpts rather than complete files. Mermaid diagrams describe ownership and flow. Callouts identify principles worth carrying into other systems. Sections

labelled as tradeoffs or limits are intentionally candid: they distinguish what FrameShift guarantees from what remains procedural, local, or future work.

A Note on Scope and Honesty

Atomic filesystem rename is not a distributed transaction. A valid PNG header is not proof of artistic correctness. At-least-once execution and delivery can repeat external effects. An optional object store adds a second durability domain. A local machine can sleep, overheat, run out of disk space, or disappear from the network. No amount of polished interface design removes those facts.

The goal of FrameShift is therefore not to pretend failure has been eliminated. The goal is to make failure explicit, bounded, observable, and recoverable. This book is an account of how that goal shaped a real system, one decision at a time.

Contents

1. [Part 1 — From an Idea to a Local-First Render Platform](#)
2. [Part 2 — Choosing the Stack: Why Go, Next.js, Tailwind, Bun, and SQLite](#)
3. [Part 3 — Designing an Authoritative Render Architecture](#)
4. [Part 4 — Treating Blender Files as Untrusted Input](#)
5. [Part 5 — Durable Scheduling: Frames, Leases, Retries, and Recovery](#)
6. [Part 6 — Building the Workers: Blender, Apple Silicon, and Real-Time Observability](#)
7. [Part 7 — From Frames to Deliverables: FFmpeg, Artifacts, and Publishing](#)
8. [Part 8 — Identity and Collaboration: Authentication, Teams, Email, and Approval](#)
9. [Part 9 — Operating FrameShift: Storage, Backups, PostgreSQL, Docker, and Deployment](#)
10. [Part 10 — Turning Infrastructure into a Product: UI, Testing, and Lessons Learned](#)

Part 1 — From an Idea to a Local-First Render Platform

FrameShift began with a practical question: what would it take to make Blender rendering feel like a managed service without first building a cloud service?

The starting environment was deliberately ordinary. There was one Mac, Blender was already installed in its standard application location, and the machine had enough CPU, memory, and storage to render useful work. The missing piece was not rendering capability. Blender could already render a frame or an animation from its own interface or from the command line. The missing piece was operational structure: a place to submit scenes, preserve settings, understand progress, recover from interruption, find outputs, and answer the deceptively difficult question, “What is the render actually doing right now?”

That distinction shaped the project. FrameShift did not need to replace Blender’s render engine. It needed to coordinate Blender reliably. It also needed to do so without assuming a Kubernetes cluster, a hosted database, an object-storage account, or a permanently connected internet service. The result is a local-first Blender render manager with a Next.js application for people, a Go backend for authority, and Go workers that discover and run Blender.

This first part of the building story is about the decisions behind that shape. It covers the original problem, the constraints that made those decisions useful, and the way a seemingly small local tool accumulated the responsibilities of a render platform.

The Problem Was Coordination, Not Pixels

The first audience for FrameShift is easy to picture: an individual Blender artist, a small studio, or a technical operator with a capable Mac that is already part workstation and part render machine. They do not necessarily need a commercial render farm. They need a clearer and safer workflow around the compute they already own.

A direct Blender render works well when a person is present and the job is simple. The workflow becomes harder as soon as a render spans many frames or needs to survive ordinary failures. The user has to remember which scene version was rendered, which camera and engine were selected, whether an output is complete, and where Blender wrote it. If the process exits halfway through, someone has to determine which frames are valid. If a scene is replaced, old outputs still need to remain attributable to the old input. If several people share the machine, permissions and approvals begin to matter. Even on one computer, rendering is a scheduling problem, a provenance problem, and a state-management problem.

FrameShift therefore treats a render as more than a child process. A render is a durable job composed of durable frame tasks, tied to an immutable scene version and resolved render

settings. It has attempts, progress, logs, outputs, finalization, and a controlled lifecycle. Blender remains the program that produces pixels, but FrameShift owns the surrounding truth.

Design callout: manage the work, do not impersonate the renderer. FrameShift delegates rendering and scene inspection to Blender. Its value comes from making that execution observable, recoverable, authorized, and repeatable.

This approach also keeps the product legible. Artists see projects, scenes, renders, and outputs rather than shell commands. Operators see workers, leases, resource limits, and failures. Administrators see storage, backups, retention, and restore plans. These are different views of the same durable system rather than separate sources of truth.

One Mac Is a Constraint and an Advantage

Building on one Mac imposed useful limits. Blender had a concrete default location:

```
BLENDER_PATH=/Applications/Blender.app/Contents/MacOS/Blender
FRAMESHIFT_WORKER_SLOTS=1
FRAMESHIFT_DATA_DIR=./var
```

Those defaults say a great deal about the initial product. There is one known Blender executable, one worker slot, and one managed runtime directory. The platform can start small enough to understand completely. A developer can run the API, worker, and web application in separate terminals, open the browser, and watch a real Blender process execute on the same host.

This constraint discouraged premature infrastructure. SQLite is the default database, so the authoritative state can live beside the application. Uploaded scenes and generated artifacts stay under `var/storage`. Mailpit supplies a local SMTP sink. The only baseline container in Compose is Mailpit; PostgreSQL and MinIO exist behind optional profiles for installations that need them. The local development stack therefore does not pretend that every user needs production topology on day one.

The Mac constraint also forced honesty about hardware. FrameShift discovers actual Blender Cycles devices, including Apple Metal devices, and records what a worker can use. It reports real host CPU, memory, disk, and worker-process telemetry. It explicitly avoids fabricating GPU-utilization numbers when macOS offers no stable supported counter. “Unavailable” is more trustworthy than a polished but invented chart.

There are tradeoffs. One machine has finite thermal capacity, memory, and disk bandwidth. A web server, database, Blender process, and finalizer may contend for the same resources. Laptop sleep, application upgrades, and reboots are normal rather than exceptional. A single worker is not highly available. These facts could have been treated as reasons to begin in the

cloud. FrameShift instead treats them as domain requirements: admission safety gates, durable tasks, expiring leases, resumable uploads, and restart-safe finalization make a local machine more dependable.

The constraint became an architectural test. If the system cannot explain and recover from work on one computer, adding ten computers will only distribute the confusion.

Local-First Does Not Mean Local-Only

“Local-first” is sometimes used as a synonym for offline UI state or peer-to-peer synchronization. FrameShift uses it in a more operational sense: the useful default deployment is owned, inspectable, and runnable on the user’s own machine. Core work does not depend on a remote control plane.

That philosophy appears throughout the repository:

- SQLite is the default, while PostgreSQL is optional.
- Managed filesystem storage is the default, while S3-compatible cold storage through MinIO is optional.
- A local worker can render directly from managed scene paths, while a network worker can download an authenticated immutable bundle.
- Mailpit makes notification flows testable without a third-party email provider.
- macOS LaunchAgents can package the server, worker, and web processes into a persistent local installation.
- Offline restore refuses to operate while the backend is reachable, favoring explicit maintenance over clever live mutation.

Local-first is therefore not a rejection of networking. FrameShift has authenticated network workers, PostgreSQL support, object storage, TLS, Prometheus metrics, and signed webhook publishing. It is a rule about defaults and dependencies. External services extend the installation; they do not grant the application permission to function.

This produces an important educational tradeoff. A hosted-only product can centralize upgrades, secrets, and observability. A local-first product gives users ownership and low-latency access to their files, but it must package operational responsibilities carefully. Paths need predictable roots. Backups must include both database and managed storage. Health checks must explain missing Blender or FFmpeg installations. Defaults need to work without silently becoming production credentials. FrameShift’s `local-development-worker-token`, for example, is convenient for development and explicitly something to replace outside it.

Local-first principle: the simplest complete FrameShift installation should be valuable on one Mac, while every extension toward a networked studio should preserve the same authoritative model.

The Go Backend Is the System of Record

FrameShift has three active participants. The Next.js frontend presents workflows and submits user intent. Workers advertise capabilities, claim tasks, execute Blender, and report measured results. The Go backend validates both sides and persists the durable outcome in SQLite or PostgreSQL.

This division is more than a technology preference. It prevents the browser and worker from becoming competing authorities.

Consider a user pressing Cancel. The browser knows that the user wants cancellation; it does not know whether a frame has just completed, whether the job is already finalizing, or whether the user still has permission. It sends a command. The backend checks the current state and capability, records the transition, updates queued frame tasks, and exposes the new truth. The worker eventually observes cancellation through the controlled protocol and stops Blender. The button is an expression of intent, not a database update disguised as a UI event.

The same rule applies in the other direction. A worker can report that Blender exited successfully and provide a checksum, dimensions, duration, and output metadata. It cannot declare the whole job complete. The backend verifies lease ownership, records the frame result, aggregates the remaining task states, and decides whether the job should continue, fail, pause, or enter finalization. Only after FFmpeg outputs have been generated, probed, hashed, and atomically promoted does the backend mark the render completed.

The domain model makes allowed movement explicit:

```
JobQueued:      {JobRunning: true, JobPaused: true, JobCanceling: true},
JobRunning:     {JobPausing: true, JobFinalizing: true, JobFailed: true,
JobCanceling: true},
JobFinalizing: {JobCompleted: true, JobFailed: true},
```

This small map is representative of a larger principle: durable state changes should be enumerated, validated, and testable. A job cannot move directly from QUEUED to COMPLETED because real work and finalization must occur between those states. A completed job is terminal because rewriting its history would make outputs and audit events ambiguous. A failed job can return to the queue because retry is a new controlled attempt, not a denial that failure occurred.

Intent Versus Authority

Separating intent from authority solves several common distributed-systems problems even before the system is distributed across several machines.

First, it handles races. Two browser tabs might issue conflicting commands. A worker might finish while cancellation is being requested. Two workers might ask for the next frame simultaneously. If each participant modifies its own idea of state, reconciliation becomes the product. FrameShift instead places the deciding operation inside a backend transaction. A frame claim updates a task only when it is still queued, binds it to one worker, increments the attempt, establishes a lease expiry, creates an attempt record, updates progress, and emits an event before committing.

A representative part of the claim logic is intentionally conditional:

```
UPDATE frame_tasks
SET status = ?, worker_id = ?, lease_expires_at = ?, attempt = ?
WHERE id = ? AND status = ?
```

The final `status = QUEUED` condition is not incidental. It is the database-level guard that makes a stale claim lose cleanly. Rows affected must equal one. The backend, rather than either worker, decides who won.

Second, the distinction enables recovery. Claims expire. At the beginning of a later claim transaction, FrameShift returns tasks with expired worker leases to the queue, or marks them canceled if their parent job is canceling. The durable record outlives the process that held it. This is why a heartbeat is evidence of worker liveness rather than proof of task completion, and why an output file appearing on disk is not by itself authoritative.

Third, it improves security. A worker token grants access to a narrow transport surface: heartbeat, claim, renew, logs, scene bundle, output upload, complete, and fail. The worker does not receive general database access. User APIs remain session-authenticated and workspace-authorized. Each actor gets enough power to report facts or request actions, but the backend retains policy decisions.

There is a cost. This model requires more states, more transaction code, and explicit conflict handling. A direct “run Blender and update the page” implementation would be shorter. It would also lose work on restart, make retries ambiguous, and allow edge cases to accumulate in UI code. FrameShift accepts backend complexity because it buys a simpler mental contract everywhere else: if it is durable and authoritative, it came through the Go backend.

Building in Milestones

The platform’s current breadth can obscure how it can be built incrementally. The productive path is not to design every studio feature before rendering the first frame. It is to establish one reliable vertical slice and then deepen the guarantees around it.

An initial milestone can be described as “prove the machine.” Discover the configured Blender executable, run it without shell interpolation, submit a known local scene, and produce one frame. This validates the fundamental dependency and exposes platform-specific behavior early.

The next milestone is “remember the work.” Introduce the Go API and SQLite schema for users, projects, jobs, workers, and durable events. Replace transient process state with a job lifecycle. A page refresh or server restart should no longer erase the meaning of the render.

Then comes “separate coordination from execution.” Move Blender launching into a worker. Add heartbeat, frame claims, lease renewal, retries, and result reporting. At this point the same contract can support a worker on the API host or a remote worker that downloads a scene bundle.

The following milestone is “trust the inputs and outputs.” Bound uploads, verify Blender container signatures, hash files, isolate storage, inspect scenes under controlled Blender startup, render into temporary directories, validate PNG dimensions, and atomically promote successful results. Add finalization only after frames themselves have a reliable identity.

User-facing richness can then grow on top of stable facts: live progress, searchable logs, estimates, calibration, notifications, scene libraries, presets, output playback, and reports. Collaboration follows with workspaces, roles, invitations, approval workflows, quotas, and audit history. Operational maturity adds backups, restore planning, retention, storage tiers, worker draining, metrics, and publishing.

This sequence illustrates a useful principle: each milestone should leave the system usable while introducing one new kind of durability. The work progresses from rendering, to remembering, to coordinating, to validating, to collaborating, and finally to operating.

Milestone test: after each increment, unplug or stop one component and ask what truth remains. The answer should increasingly be “enough to recover safely.”

A Threat Model for Creative Files

The most important security assumption is that uploaded `.blend` files and scene archives are untrusted. Creative files are not harmless merely because they are familiar to artists. A Blender file may contain embedded scripts. An archive may contain traversal paths, absolute paths, backslashes intended to confuse platform handling, symbolic links, too many entries, or highly compressed data designed to expand far beyond its upload size.

FrameShift’s process boundary addresses the first risk. Blender is launched with an argument array through `exec.CommandContext`, not through an interpolated shell command. Rendering uses controlled flags:

```
exec.CommandContext(ctx, blenderPath,  
    "--background", "--factory-startup", "--disable-autoexec", scenePath,  
)
```

`--disable-autoexec` prevents automatic execution of scripts embedded in uploaded scenes. `--factory-startup` reduces dependence on a user's mutable Blender preferences. An argument array prevents filenames or settings from turning into shell syntax. FrameShift still uses a controlled Python expression where Blender's API is needed for inspection or render overrides, but that expression is generated by FrameShift rather than sourced from the upload.

Archive handling adds another defensive layer. FrameShift permits at most 10,000 entries, caps declared expansion at four times the configured upload limit, rejects compression ratios above 100:1, rejects symbolic links, and validates every archive path. A package must contain exactly one `.blend` file. Present assets are normalized, typed, checksummed, and stored through checksum-addressed blobs; missing and external references are recorded rather than quietly assumed to exist.

This threat model extends to outputs. Blender writes a frame into a per-attempt temporary directory. FrameShift opens the result as a PNG, validates it, calculates its SHA-256 checksum, applies restrictive permissions, and only then renames it into the durable output location. FFmpeg finalization follows the same philosophy: create a complete temporary artifact set, validate the H.264 output with FFprobe, hash the files, and promote the set atomically.

The tradeoff is deliberate friction. Some Blender workflows rely on automatic scripts, custom startup state, or external paths. FrameShift refuses to make those behaviors implicit because a shared render service changes the trust boundary. Supporting a workflow is not the same as executing everything a file requests. Explicit, controlled capabilities can be added over time; automatic execution cannot be safely taken back after it has become an assumption.

Product Principles Hidden in the Architecture

Several product principles emerge from these technical choices.

Real facts beat optimistic presentation. Progress comes from durable frame state and parsed Blender sample output. Estimates come from successful historical attempts normalized by engine, effective pixel count, and sample ratio. When matching history does not exist, FrameShift asks for calibration instead of inventing an ETA. Hardware views report available devices without claiming unsupported utilization metrics.

Immutable inputs make results explainable. Scene replacement creates a new version. Existing jobs remain bound to the version they used. New scene versions begin as drafts, production jobs bind approved versions, and calibration can run before approval. This gives experimentation room without weakening the provenance of production output.

Partial success is not completion. A directory of frames is useful, but a render configured to produce an MP4 is not complete until finalization succeeds. FrameShift has a distinct `FINALIZING` state and a durable finalization lease. Completion means the intended artifact set, including preview, thumbnail, frame archive, report, and manifest, was created and promoted together.

Failure should preserve evidence. Render attempts store duration, exit code, peak memory, peak multicore CPU, and bounded structured logs. Failed leases and retries remain visible. Notifications and email delivery use durable outboxes, so an SMTP outage does not roll back a completed render.

Local resources deserve policy. CPU, RAM, disk utilization, and free-space reserves are checked before job admission and worker claims. Workspace storage quotas and concurrent-render limits add a product-level boundary above host safety. A queue should not merely accept work; it should know when beginning that work would endanger the machine.

Operations are part of the product. The oldest local account becomes the machine administrator. Backups combine managed storage with an online SQLite snapshot or PostgreSQL dump. Restore is offline and refuses to run while the backend is reachable. Workers can drain before removal. Retention and hot/cold tiers are explicit. These features acknowledge that a local-first application still owns valuable state.

How the Requirements Evolved

FrameShift's requirements expanded through pressure from real boundaries rather than from a desire to collect features.

Once a scene could be uploaded, upload size and restart behavior mattered, leading to bounded streaming and resumable sessions with authenticated offsets, checksums, expiry, cancellation, and quota admission with completion-time rechecking. Once a scene could reference textures, a `.blend` file alone was insufficient, leading to hardened ZIP packages, typed asset manifests, shared checksum blobs, and immutable version comparison.

Once a worker could render one frame, an animation required per-frame tasks. Per-frame tasks raised questions about duplicate claims, worker crashes, retry limits, and cancellation. Those questions led to leases, renewal, stale recovery, attempts, and frame-level commands. Long frames made job-level percentages inadequate, so Blender log parsing, phase reporting, sample progress, and continuously recalculated live estimates followed.

Once all frames could finish, users needed something convenient to watch and download. That introduced durable FFmpeg finalization, FFprobe validation, checksums, manifests, previews, thumbnails, ZIP archives, and reports. Once outputs became valuable, authorization, retention, pinning, cold storage, publishing approval, and backups became necessary.

Once more than one person could use the system, a project owner was no longer enough. Personal and organization workspaces, invitations, roles, capability enforcement, ownership transfer, audit history, storage quotas, and concurrent-render limits established collaboration boundaries. Once scene changes could affect production, draft/review/approved/rejected states and reviewer assignments made the input lifecycle explicit.

Once workers could move across the network, local file paths stopped being a sufficient transport. Immutable scene bundles, authenticated result upload, server-side PNG validation, device-compatible routing, worker slots, and drain controls preserved the same authority model across machines.

This evolution is instructive because each new requirement can be traced to a previous capability becoming valuable enough to protect. Resumable upload protects large inputs. Leases protect distributed work. Approval protects production provenance. Backups protect accumulated history. The architecture did not become platform-like because “platform” was the initial label; it became a platform because reliable rendering touches identity, scheduling, storage, security, observability, and operations.

Choosing Complexity Deliberately

FrameShift could have used a hosted queue, cloud object storage, and a managed identity provider from the beginning. Those services solve real problems. They also change the learning surface and the ownership model. A local SQLite transaction makes the relationship between a frame claim and its event directly inspectable. A managed storage root makes atomic rename and backup boundaries concrete. Running Blender with `exec.CommandContext` exposes cancellation and process metrics without an additional remote execution layer.

Conversely, local-first does not justify reinventing everything. Blender remains the rendering engine. FFmpeg and FFprobe handle video production and validation. Next.js supplies the browser application framework. PostgreSQL and S3-compatible storage are available when deployment needs outgrow the defaults. The design goal is not maximal self-sufficiency; it is a coherent center of authority with replaceable edges.

Go is a good fit for that center. Its standard library provides HTTP serving, contexts, process execution, hashing, archive reading, and filesystem operations. Goroutines fit heartbeat, delivery, finalization, and worker loops. Static binaries support the macOS installation story. Most importantly, the backend can keep policy close to transactions without dividing core state decisions across several services.

The cost is that the monolith carries many responsibilities. That is acceptable while those responsibilities share invariants. Job completion, notifications, events, quotas, and artifact records often need one transactional view. Splitting them too early would replace function calls

and database transactions with protocols and eventual consistency. If scale eventually requires separation, the durable tables and explicit leases already identify useful boundaries.

The Shape of the Platform

The system that emerges is still recognizable from the initial Mac-based idea. The browser runs at port 3000, the Go API at 8080, Mailpit at 8025, and a local worker finds Blender under `/Applications`. SQLite and `var/storage` keep the default installation tangible. A user can create a project, upload and inspect a scene, approve a version, choose render settings, submit frames, watch real progress, and retrieve validated artifacts.

Around that simple path is a set of guarantees: uploads are untrusted, jobs are durable, transitions are validated, claims expire, attempts are recorded, outputs are promoted atomically, estimates admit uncertainty, and external services remain optional. Those guarantees are what turn a command runner into a render platform.

The central lesson is not that every local tool needs this much machinery. It is that requirements should follow the value of the state being protected. The first successful frame proves that rendering is possible. The next question is whether the system can explain, repeat, and recover that success. FrameShift's architecture is the accumulating answer.

With the product constraints established, the next question is which technologies should own each responsibility.

Part 2 — Choosing the Stack: Why Go, Next.js, Tailwind, Bun, and SQLite

FrameShift is a local-first render manager, not merely a web interface around a command. It accepts untrusted Blender scenes, records durable jobs, coordinates workers, launches expensive native processes, recovers leases after failures, finalizes image sequences, and presents the resulting state to a browser. Those responsibilities make stack selection an exercise in assigning authority and failure boundaries.

The central rule is simple: the Go backend is the system of record. The browser sends user intent, and workers report observations and results, but neither decides the durable truth of a job. If a worker says that frame 42 finished, the backend still verifies that the report belongs to a valid lease and applies the legal state transition in a transaction. If the browser asks to cancel a render, the backend decides what can be canceled and records the outcome. This rule influenced every major technology choice.

The selected stack is deliberately asymmetric:

Concern	Primary choice	Reason it fits the boundary
Authoritative API and scheduler	Go and <code>net/http</code>	Explicit concurrency, strong process control, simple native deployment
Durable state	SQLite through <code>database/sql</code>	Zero-service local setup and transactional authority
Multi-host database path	PostgreSQL through <code>pgx</code>	Better concurrent writes and operations when deployment grows
Product interface	Next.js App Router and React	Routing, layouts, server/client boundaries, and production builds in one framework
Styling	Tailwind plus custom CSS	Fast composition without giving up domain-specific visual rules
Frontend tooling	Bun	Fast installs and one consistent script runner
Render engines and media tools	Native Blender, FFmpeg, and FFprobe	Use the mature domain executables directly
Optional infrastructure	Docker Compose	Reproducible services without containerizing hardware-sensitive core work

No entry is universally best. Each is the result of FrameShift's decision criteria: correctness after restart, safe native-process execution, low local setup cost, inspectable behavior, Apple Silicon support, and a credible path from one workstation to a small network of render workers.

Why Go Owns Authority

FrameShift needs concurrency, but concurrency is not itself the hardest problem. The hard problem is coordinating concurrent work while preserving one authoritative history. Worker heartbeats, frame claims, lease renewals, log batches, event streams, finalization claims, notification delivery, and cleanup can all overlap. At the same time, cancellation must reach child processes promptly and shutdown must not abandon state casually.

Go is a strong fit because its concurrency model is present without dominating the program. A goroutine can run each worker slot or background delivery loop, channels can communicate completion and cancellation, and `context.Context` carries deadlines across HTTP, database, and process boundaries. The renderer can start Blender, observe process metrics on a ticker, and react to either process completion or context cancellation with ordinary control flow:

```
command := exec.CommandContext(ctx, blenderPath, args...)
done := make(chan error, 1)
```

```
go func() { done <- command.Wait() }()

select {
case err := <-done:
    // Validate and report the result.
case <-ctx.Done():
    // CommandContext terminates the child process.
}
```

That is useful operationally. The same cancellation signal that expires an HTTP request or render lease can stop native work. Go's race detector, conventional table-driven tests, and static compilation also suit scheduler code where a subtle shared-state bug can become a duplicate render or an invalid transition.

Deployment matters just as much. The backend, worker, and administration utility compile into separate native binaries. A local installation does not require a language runtime, virtual environment, or dependency tree beside each process. On macOS they can run as LaunchAgents, and a remote worker can receive the worker binary plus configuration. The frontend still has its own runtime needs, but the processes that own job state and control Blender remain straightforward native programs.

Why not Node.js for everything?

Node.js would reduce the number of languages and share TypeScript types with the browser. Its HTTP ecosystem is mature, and asynchronous I/O is excellent for an API. It was not selected for the authoritative layer because FrameShift is also a process supervisor and scheduler. Long-running child processes, per-slot loops, cancellation, telemetry sampling, and graceful shutdown are possible in Node, but they tend to combine event-loop concerns, promise cancellation conventions, worker threads, and package-specific lifecycle behavior.

That does not make Node unreliable. It means the cost model is less favorable here. FrameShift would trade Go's direct goroutine/context/process model and native binaries for shared language syntax. Shared syntax is useful, but it does not remove the network contract or the need to validate untrusted worker messages. The backend API remains the real contract either way.

Why not Python?

Python is compelling around Blender because Blender embeds Python and most Blender automation examples use `bpy`. A Python service could share scripts and data shapes with in-process Blender code. It would also offer excellent media and scientific libraries.

The distinction is where Python runs. FrameShift intentionally uses small, controlled Python expressions inside Blender when Blender's API is required, but it does not make an uploaded

scene's scripts authoritative or automatically executable. For the server and worker supervisor, Python would introduce interpreter and environment management, packaging choices, and more care around CPU-bound concurrency and process models. Async Python can handle network I/O well, and multiprocessing can use multiple cores, but the combined operational model is more complicated than Go's for this workload.

Python remains an appropriate alternative for an analysis service or a Blender-specific plugin. It is less attractive as the single durable scheduler, API, worker supervisor, and distribution artifact.

Why not Rust?

Rust offers excellent performance, memory safety, native binaries, and precise control over concurrency. It could implement FrameShift successfully. The deciding cost is development friction relative to the risk being addressed. Most of FrameShift's heavy computation happens in Blender and FFmpeg, not in the orchestration process. The coordinator benefits more from clear transactions, cancellation, and maintainable state-machine code than from eliminating garbage collection or optimizing every allocation.

Rust's ownership model and async ecosystem would buy stronger compile-time guarantees, but at a higher implementation and onboarding cost. Go provides enough safety and performance while keeping HTTP handlers, SQL transactions, and process loops readable. Rust would become more attractive if FrameShift acquired a large in-process media pipeline, hard real-time constraints, or memory-safety exposure through native libraries.

Backend option	Main advantage	Main cost for FrameShift	Decision
Go	Simple concurrency, contexts, native binaries	Separate language from frontend	Best balance for authority and workers
Node.js/TypeScript	One language across the product	More runtime and lifecycle machinery around native workers	Keep on the UI side
Python	Blender familiarity and rich automation ecosystem	Environment packaging and mixed concurrency models	Use only at controlled Blender boundaries
Rust	Maximum native safety and control	Higher development cost for mostly I/O orchestration	Reconsider for performance-critical native components

Standard net/http , Not a Large Web Framework

Go's standard HTTP stack now provides method-aware route patterns such as `GET /healthz` and paths with variables such as `/api/v1/jobs/{id}`. `FrameShift` registers these on `http.ServeMux` and wraps handlers with explicit authentication, worker authorization, system-administrator checks, and common middleware.

This choice keeps the request path visible. A reader can follow route registration, middleware, handler, store method, and transaction without learning a framework's container, decorators, or implicit binding rules. It also reduces dependency churn in the most security-sensitive process. Standard types such as `http.Handler`, `http.Request`, and `http.ResponseWriter` compose easily in tests and remain compatible with observability or middleware packages later.

The cost is real. `FrameShift` must provide its own JSON error conventions, request decoding, validation, middleware ordering, and route organization. A framework such as `Gin`, `Echo`, or `Fiber` could reduce boilerplate and offer convenient binding. That trade is worthwhile for teams optimizing rapid CRUD delivery or standardizing many services. In `FrameShift`, most complexity lies in domain transitions and storage rules rather than HTTP syntax, so hiding a few lines of handler code would not remove the difficult work.

The criterion is not "standard library at all costs." It is whether a dependency owns enough genuine complexity to justify another abstraction. The router does not. The `MinIO` client and `PostgreSQL` driver do.

Explicit SQL and Visible Transactions

`FrameShift` uses `database/sql` with explicit queries rather than an object-relational mapper. Render scheduling depends on exact transactional behavior: atomically claim one eligible task, renew the correct lease, reject stale results, append audit events, and update job aggregates without allowing impossible states. SQL expresses those operations closer to the database's actual concurrency model.

A typical boundary looks conceptually like this:

```
tx, err := db.BeginTx(ctx, nil)
if err != nil {
    return err
}
defer tx.Rollback()

// Read eligibility and apply the state transition in this transaction.

return tx.Commit()
```

Explicit SQL makes query shape, lock scope, and transaction duration reviewable. It avoids accidental query waterfalls and does not require domain models to mirror table rows. That separation matters: a render job is a state machine with permissions and recovery semantics, not just a record with generated `Save` methods.

The project still adds a small adapter around `database/sql`. Application queries use `?` placeholders, and the adapter translates them to PostgreSQL's `$1`, `$2`, and so on while respecting quoted strings. Schema setup also accounts for the limited type differences used by the project. This is intentionally much smaller than pretending SQLite and PostgreSQL are identical. Database-specific behavior must still be tested, especially around timestamps, generated IDs, migrations, isolation, and concurrent claims.

An ORM would improve schema navigation, relation loading, and routine CRUD generation. It could also provide typed query construction. Its cost is another model layer and the temptation to express scheduler transactions through object operations whose SQL is less obvious. A typed SQL generator would be a reasonable future compromise if query volume makes manual scanning or result mapping error-prone. For now, direct SQL keeps authority legible.

SQLite First, PostgreSQL Later

SQLite matches FrameShift's local-first promise. A new installation can create a database file under managed local state and immediately gain transactions, indexes, constraints, and crash recovery. There is no database daemon to install, secure, start, upgrade, or back up separately. The database and storage tree can be treated as one workstation-owned system, which makes development, demos, and single-machine studio use much easier.

FrameShift uses modernc.org/sqlite, a CGo-free SQLite implementation for Go. Avoiding CGo is particularly valuable for distribution: builds do not need a local C compiler or platform-specific SQLite linkage, and cross-compilation remains simpler. On macOS and Apple Silicon, that removes a class of architecture mismatches involving `x86_64` libraries, `arm64` binaries, and linker configuration.

SQLite's limitations are equally important. It permits many readers but serializes writes. A busy scheduler with many network workers, frequent heartbeats, streamed logs, notifications, and API mutations can create write contention even when every individual transaction is short. A database file also does not become a safe network database by placing it on a shared filesystem. Operational tools, replication, role management, and high-availability patterns are much richer around PostgreSQL.

The migration criterion is workload and topology, not prestige. PostgreSQL becomes the better default when one or more of these conditions hold:

- multiple server instances need to coordinate against the same database;

- sustained worker and user traffic creates measurable SQLite write contention;
- backups need database-native operational tooling and remote storage separation;
- the installation requires managed database availability, access controls, or observability;
- network deployment makes a workstation-local database file the wrong ownership model.

The PostgreSQL path uses `github.com/jackc/pgx/v5/stdlib`, allowing the application to retain the `database/sql` boundary while using `pgx` as the driver. `Pgx` is the conventional high-quality Go PostgreSQL implementation, supports modern PostgreSQL behavior, and leaves room to adopt `pgx`-native features later if a demonstrated need outweighs portability.

Supporting both databases has a cost: migrations and queries must stay inside a tested common subset or branch deliberately by dialect. Placeholder translation and broad schema substitutions are helpful, but they are not a substitute for integration tests against PostgreSQL. Every new query should be evaluated for both engines. The payoff is a smooth product progression: zero-service setup first, service-grade concurrency when the installation actually needs it.

Next.js App Router Instead of Plain React

The browser is not authoritative, but it is a substantial product surface. `FrameShift` has public authentication pages, an authenticated dashboard, workspace and project navigation, scene review, render controls, live progress, output playback, administration, and responsive behavior. Plain React supplies components and state, but it does not choose routing, layouts, bundling, code splitting, development serving, production serving, metadata, fonts, or server/client rendering boundaries.

Next.js App Router supplies those decisions as one coherent framework. Directory-based routes make product areas discoverable. Nested layouts can preserve shared navigation while pages change. Server Components are available for work that does not need browser interactivity, while client components can own uploads, polling, dialogs, and live controls. Route-level loading and error boundaries provide a natural way to contain slow or failed sections.

Even though the Go API owns data and authorization, Next.js remains useful. It is not being selected as a second backend. It is the web delivery framework. The browser still treats API responses as non-authoritative until the Go service accepts and persists an action. This avoids splitting job rules between server actions and Go handlers.

A plain React application built with Vite would be lighter and very capable. It would reduce framework conventions and work well for a fully client-rendered dashboard. `FrameShift` would then need to select and integrate a router, layout conventions, document metadata, font loading, production hosting, and any server rendering strategy. That can be the right choice for an embedded control panel or a small single-page tool. `FrameShift` already has public and

authenticated route groups and benefits from a framework-level application structure, so App Router earns its weight.

The costs are framework upgrades, caching semantics, distinctions between server and client components, and a larger mental model than React alone. The project controls those costs by keeping business authority in Go and using Next.js for composition and delivery rather than duplicating scheduler logic.

Tailwind Alongside Custom CSS

Tailwind is useful for local composition: flex and grid layouts, spacing, responsive changes, typography, and state variants can be read beside a component. It speeds iteration on dense operational screens because a developer can adjust structure without inventing a class name for every wrapper. Tailwind 4's CSS-first integration also keeps setup small through `@import "tailwindcss"` and the PostCSS plugin.

FrameShift does not treat utilities as the entire design system. Its global stylesheet defines semantic color variables such as canvas, paper, ink, line, status colors, and navigation colors. It also contains stable component rules for the sidebar, navigation items, workspace switcher, and other repeated product surfaces. The root layout loads Instrument Sans and JetBrains Mono through `next/font`, exposing them as CSS variables.

This hybrid is deliberate:

```
:root {
  --canvas: #f4f3ef;
  --ink: #20211e;
  --coral: #c9472d;
}

.nav-item {
  min-height: 42px;
  border-radius: 6px;
}
```

Utilities answer "how is this instance arranged?" Custom CSS answers "what is a FrameShift navigation item?" and "which visual tokens define the product?" Stable dimensions and named states are especially important in a render dashboard, where counters, progress, logs, and changing statuses must not shift the interface unexpectedly.

The cost of mixing approaches is possible inconsistency. A component can bypass a semantic token with an arbitrary color, or custom CSS can duplicate utility behavior. Review criteria keep the boundary useful: prefer utilities for one-off layout and responsive composition; prefer

custom classes and variables for repeated identity, complex state styling, and rules that must remain stable across components.

CSS Modules, styled-components, or a full component system were viable alternatives. CSS Modules provide excellent local scoping but still require many names for layout wrappers. Runtime CSS-in-JS adds machinery that is difficult to justify for this operational interface. A large component kit would speed common controls but could impose a visual language and dependency surface larger than the product currently needs.

Why Bun Runs the Frontend Toolchain

Bun is the frontend package manager and script runner. For FrameShift, its value is practical: fast dependency installation, a compact lockfile workflow, and quick execution of Next.js, ESLint, and Playwright scripts. The Makefile can expose one predictable command surface:

```
cd apps/web && bun install
cd apps/web && bun run lint
cd apps/web && bun run build
```

Bun does not change the browser runtime, and FrameShift does not depend on replacing the Go server with Bun. This limited role reduces risk. Next.js and the selected tooling still operate through their normal package interfaces.

Node with npm, pnpm, or Yarn would all work. Pnpm is particularly credible for a larger monorepo because of its strict, space-efficient dependency model. Bun was chosen because this repository has one frontend package and values fast local setup over advanced workspace management. The costs are a younger runtime, occasional compatibility differences in install scripts or Node APIs, and another prerequisite for contributors. The package manifest therefore pins the package-manager version and explicitly controls trusted install scripts for packages that need them.

The decision should be revisited if Bun compatibility becomes a recurring source of exceptions, if a large multi-package frontend monorepo develops, or if deployment standardizes on a Node-only platform. Tool speed is valuable only while behavior remains boring and repeatable.

Native Blender and FFmpeg

Blender and FFmpeg are not libraries that FrameShift should reimplement. They are mature native applications with optimized codecs, render engines, hardware integration, and stable command-line contracts. FrameShift's responsibility is to invoke them safely, constrain their inputs, observe them, validate their outputs, and record authoritative outcomes.

Blender runs as a native child process with an argument array passed to `exec.CommandContext`; no shell command is interpolated. Rendering uses background and factory-startup modes and includes `--disable-autoexec`, so scripts embedded in uploaded scenes are not automatically trusted. Output first lands in an isolated temporary directory. `FrameShift` then decodes the PNG header, checks dimensions, applies permissions, and atomically promotes the file. This process boundary isolates Blender crashes and allows hard cancellation without placing Blender's large native runtime inside the server.

`FFmpeg` converts completed frame sequences into H.264 MP4 outputs, previews, and thumbnails. `FFprobe` independently inspects the generated media before completion is accepted. This validation step matters because a zero exit code alone is not a sufficient artifact contract. The finalizer can check codec, dimensions, duration, and stream presence, hash artifacts, and only then mark the durable job complete.

Library bindings were considered implicitly and rejected for the main path. Linking `libavcodec` directly would add `CGo`, platform-specific build complexity, and memory-safety boundaries while providing little benefit to an orchestration service. Running Blender as an imported Python module would tightly couple the worker to Blender's embedded runtime and make crash isolation harder. Native executables keep domain complexity behind explicit, testable process contracts.

macOS and Apple Silicon

The first deployment target changes several otherwise abstract choices. Blender's standard macOS executable lives inside its application bundle, commonly at `/Applications/Blender.app/Contents/MacOS/Blender`. `LaunchAgents` provide a native way to keep the server, worker, and web process running for a signed-in user. `Go` can produce arm64 binaries directly, while the `CGo`-free SQLite driver avoids requiring an arm64 compiler and native SQLite library during normal builds.

Apple Silicon also affects rendering. Blender Cycles can expose Metal devices, and `FrameShift` discovers those devices through a controlled Blender invocation rather than assuming CUDA-style hardware. CPU remains a safe default; an explicit device selection is persisted and applied only where Cycles supports it. The system must not invent GPU utilization when macOS does not expose a stable supported counter. Device availability, render configuration, and utilization are different facts.

Native execution avoids another common trap: running Blender in an amd64 Linux container under emulation on an arm64 Mac. That path can lose direct Metal access, complicate application-bundle discovery, increase filesystem transfer costs, and produce performance numbers unrelated to native rendering. `FFmpeg` installed through Homebrew similarly provides an architecture-appropriate executable and codec set.

Cross-platform libraries still matter. `github.com/shirou/gopsutil/v4` provides CPU, memory, disk, and process telemetry behind OS-specific implementations. FrameShift uses it both for host safety gates and Blender child-process measurements such as peak resident memory and multicore CPU percentage. The library avoids hand-maintaining Darwin, Linux, and Windows system calls. Its cost is that metric availability and semantics vary by platform, so the product reports unsupported values honestly rather than fabricating parity.

Focused Library Choices

The dependency list is small because each selected library should own a difficult compatibility boundary:

Package or tool	What it owns	Why not build it locally?
<code>modernc.org/sqlite</code>	CGo-free SQLite driver	SQL engine and platform distribution are solved problems
<code>github.com/jackc/pgx/v5</code>	PostgreSQL protocol and driver behavior	Authentication, types, and wire compatibility require specialist maintenance
<code>github.com/shirou/gopsutil/v4</code>	Cross-platform host/process metrics	OS telemetry APIs differ substantially
<code>github.com/minio/minio-go/v7</code>	S3-compatible object operations	Signing, retries, multipart behavior, and compatibility are subtle
<code>lucide-react</code>	Consistent interface icons	Familiar controls should not use bespoke SVGs
Playwright	Real-browser end-to-end testing	Browser automation, waiting, tracing, and isolation are complex

The MinIO Go client supports optional S3-compatible cold storage. MinIO is useful locally because the same API shape can later target an external compatible object store, while the server retains authority over lifecycle decisions. The client does not make object storage mandatory; local managed storage remains the simplest default.

Lucide gives the React interface a coherent icon vocabulary for actions and navigation. It reduces hand-drawn SVG duplication and improves recognition, but icons still need accessible labels or tooltips when meaning is not obvious. A small icon dependency is preferable to importing an entire visual component system solely for symbols.

Playwright exercises the real Next.js application in Chromium-class browser behavior, including authentication recovery, responsive desktop and mobile paths, uploads, and accessibility checks. @axe-core/playwright adds automated accessibility assertions. These tests cost more time than unit tests and can fail for environmental reasons, so they belong above fast Go tests, linting, and builds rather than replacing them. Their value is catching integration failures that component tests cannot: incorrect routing, cookie behavior, focus management, API wiring, and viewport-specific overlap.

Docker Only for Optional Services

Docker Compose defines Mailpit, PostgreSQL, and MinIO. These are infrastructure services with stable network interfaces, persistent named volumes, and no need for direct access to the Mac's Blender installation or Metal devices. PostgreSQL and MinIO use profiles, so they start only when that deployment path is being exercised. Mailpit provides a disposable local SMTP server and inbox without making real email delivery part of development.

The core server and worker intentionally run natively. Containerizing them would complicate access to /Applications/Blender.app , GPU backends, local project files, output permissions, process telemetry, and cancellation. Mounting the Docker socket or host application directories into a privileged worker would enlarge the security boundary. On macOS, Docker runs Linux containers inside a virtual machine, so "local" paths and processes are not actually the host's paths and processes.

This is not an anti-container position. A production Go API without local Blender duties could be packaged as a container, and remote Linux workers could use purpose-built images when GPU drivers and Blender versions are controlled. The present criterion is that containers should simplify a service boundary. They simplify PostgreSQL, object storage, and test email. They do not simplify a Mac worker that must supervise native Blender accurately.

Costs, Revisit Triggers, and the Overall Rule

Every choice creates maintenance obligations. Go and TypeScript require two language toolchains. Supporting SQLite and PostgreSQL doubles part of the persistence test matrix. Next.js adds framework semantics. Tailwind plus custom CSS needs discipline. Bun adds a non-default package manager. Native Blender and FFmpeg make installation checks necessary. Docker remains a prerequisite for optional local infrastructure.

Those costs are accepted because they align with clear ownership boundaries. The stack should be reconsidered when evidence changes:

Evidence	Likely response
SQLite write waits affect claims or heartbeats	Promote PostgreSQL for that installation

Evidence	Likely response
A Go handler layer accumulates repetitive validated CRUD	Evaluate a narrow router or validation package
SQL result mapping causes recurring defects	Evaluate generated typed queries without hiding transactions
Bun compatibility exceptions outnumber speed benefits	Move frontend scripts to pnpm or npm
Next.js server features remain unused while upgrades dominate work	Re-evaluate a Vite-based client application
In-process media transformation becomes a measured bottleneck	Consider a dedicated Rust/native media component
Containerized Linux workers become the normal target	Build explicit worker images and hardware qualification tests

The most important decision is not any individual tool. It is refusing to let convenience blur authority. Go owns durable decisions. SQL transactions make those decisions atomic. SQLite makes the first installation easy, while PostgreSQL provides a measured growth path. Next.js renders and organizes the user experience without becoming a second scheduler. Workers execute Blender and report facts. Native tools do the specialist media work. Docker supplies optional services where isolation is genuinely helpful.

That division makes FrameShift understandable under failure. A browser can refresh, a worker can disappear, Blender can crash, FFmpeg can reject an output, and an optional service can be absent without changing which component decides the truth. The chosen stack is successful when those boundaries remain more obvious than the technologies themselves.

Those stack choices matter only if their authority boundaries remain explicit, so the architecture now moves from tools to invariants.

Part 3 — Designing an Authoritative Render Architecture

Render orchestration is a coordination problem disguised as a media problem. Blender consumes a scene and produces frames, but a production system must also answer harder questions: Which scene revision did this job use? Who was allowed to start it? Can two workers render the same frame? What happens if a worker disappears after writing a file but before reporting success? Can a browser reconnect without inventing state from stale notifications?

FrameShift answers those questions by making the Go backend and its database the system of record. The web application sends user intent. Workers report capabilities, progress, and results. Background loops deliver email, finalize frames, publish artifacts, and recover expired

leases. None of those participants independently decides the durable state of a job. State becomes authoritative only through a validated store operation committed to the database.

This chapter explains that design, the package boundaries that support it, and the tradeoffs involved in combining SQL transactions with large files and long-running external processes.

The authority rule

The central rule is simple:

A process may observe work or propose a transition, but only the Go store may commit authoritative application state.

That rule prevents several tempting but dangerous alternatives. The browser cannot mark a job complete because it received progress for the last frame. A worker cannot assume ownership because it found an item in an in-memory queue. The existence of `frame_000042.png` does not prove that frame 42 is complete. Even an SSE message is not the source of truth; it is a durable event projected from a transaction that already changed the source of truth.

Display Mermaid diagrams in this vault?

Only allow if you trust this vault's contents.

Allow

flowchart LR

```
Browser[Next.js app] -->|authenticated intent| API[Go HTTP API]
Worker[Render worker] -->|heartbeat, claim, renew, result| API
API --> Store[internal/store]
Store --> DB[(SQLite or PostgreSQL)]
Store --> Events[(events table)]
Events -->|SSE projection| Browser
Store -->|lease with frozen source| Worker
Worker -->|Blender output| Files[(Managed storage)]
Finalizer[Finalizer / outbox loops] --> Store
Finalizer --> Files
```

The database therefore records identities, membership, scene versions, job sources, resolved configuration, frame tasks, attempts, leases, controls, finalization work, artifacts, notifications, and events. Managed storage contains large byte sequences that SQL should not contain: uploaded `.blend` files, packaged assets, rendered frames, previews, reports, and archives. Database rows give those bytes meaning.

Boundaries in the repository

FrameShift uses Go's `internal` mechanism and a conventional command layout to make architectural ownership visible.

cmd : composition and process lifecycles

Each directory under `cmd` builds an executable:

- `cmd/server` opens the database, resolves configuration, constructs services, starts HTTP, and runs recovery, email, publishing, and finalization loops.
- `cmd/worker` heartbeats, claims leased frame tasks, invokes the renderer, renews leases, and reports validated results.
- `cmd/admin` contains operational entry points such as diagnostics and restore workflows.

Commands are composition roots. They decide which concrete services run and how environment variables are interpreted, but they should not become an alternate domain layer. For example, the server loop may call `ClaimFinalization` and then run FFmpeg, but the store decides whether finalization is claimable and whether its completion can be committed.

internal : authoritative behavior and infrastructure

The packages under `internal` divide backend responsibilities:

- `domain` defines shared vocabulary: jobs, frame tasks, scene versions, workspace roles, capabilities, telemetry, events, artifacts, and status values.
- `store` owns persistence, transactions, authorization queries, scheduling, leases, migrations, events, and durable state transitions.
- `api` authenticates HTTP requests, validates transport input, checks capabilities, and translates outcomes into HTTP responses.
- `scenes` ingests untrusted scenes and archives into managed storage, computes checksums, extracts metadata, and disables Blender auto-execution during inspection.
- `render` safely invokes Blender for a leased task.
- `finalizer` transforms completed frames into videos, previews, thumbnails, archives, reports, and manifests.
- `auth`, `hardware`, `backup`, `notifications`, and `objectstore` isolate their respective concerns.

These boundaries are not absolute layers. The important direction is that infrastructure cannot bypass store invariants. A renderer produces evidence; it does not update a `Job` struct in shared memory. A finalizer produces artifacts; `CompleteFinalization` records those artifacts and the completed job state together.

apps : clients, not authorities

`apps/web` is a Next.js App Router application managed with Bun. It presents data and issues commands through `/api/v1`. Its local React state improves responsiveness, but it is a cache

of server state. After a refresh or SSE reconnect, the application must be able to reconstruct the same truth from API reads and persisted events.

This gives the frontend freedom to use optimistic interaction carefully without granting it ownership. A Pause button may immediately show that the request is being sent, but the durable status is the status returned by the backend. The distinction matters when commands race, permissions change, or a job has already advanced.

Domain models are vocabulary, not mutable truth

`internal/domain/models.go` defines plain Go structures shared across store, API, worker, and supporting services. These types make contracts explicit without turning an in-memory object graph into the database.

For example, a job combines identity and aggregate status with immutable or resolved execution inputs:

```
type Job struct {
    ID             string
    ProjectID      string
    SceneID        string
    Status         JobStatus
    FrameStart     int
    FrameEnd       int
    CompletedFrames int
    Settings       RenderSettings
    Priority        int
    NotBefore      *time.Time
}
```

The actual representation is normalized across `render_jobs`, `job_sources`, `job_configs`, `job_kinds`, `job_scheduling`, `frame_tasks`, and related tables. That separation is useful because each concept has a different lifecycle. Scheduling priority can be indexed independently. A source revision remains frozen. Attempts accumulate while a frame task retains one stable identity.

The domain package also declares `JobStatus`, `FrameStatus`, roles, capabilities, and `CanTransition`. This keeps the language consistent. It does not mean any caller may assign those values. Store methods still load the current row, validate the requested operation, update related rows, append an event, and commit.

An architectural lesson follows: domain types describe valid concepts; transaction boundaries decide when those concepts become facts.

Transactions protect invariants

Most meaningful store operations use the same pattern:

```
tx, err := s.db.BeginTx(ctx, nil)
if err != nil {
    return err
}
defer tx.Rollback()

// Read current state, validate, and update all related rows.
// Insert the event through the same transaction.

return tx.Commit()
```

The deferred rollback makes every early return safe. `Commit` is the single publication point. This is more than tidy error handling: it prevents observers from seeing half a decision.

Job creation is a representative transaction. `createJob`:

1. Selects an accessible scene version. Production renders choose the latest approved version; calibration uses the latest version.
2. Validates the requested frame range against stored scene metadata.
3. Resolves overrides into a complete `RenderSettings` snapshot.
4. Inserts the job, frozen source, configuration, kind, scheduling row, and finalization row.
5. Inserts one frame task for every selected frame.
6. Appends `job.queued` or `calibration.queued`.
7. Commits everything together.

If frame task 87 cannot be inserted, there is no authoritative job with only 86 tasks. If event encoding fails, the state mutation also rolls back. Consumers never need to infer whether an absent event means a committed job was missed during a process crash.

Transactions also make aggregate transitions coherent. When a frame completes, the store verifies lease ownership, records output metadata, closes the attempt, stores metrics and logs, recomputes frame counts, updates the job, and possibly queues finalization. Those writes describe one logical fact and belong in one commit.

State machines make illegal states explicit

Render work lasts much longer than an HTTP request, so status cannot be a casual label. `FrameShift` models job and frame state machines.

Display Mermaid diagrams in this vault?

Only allow if you trust this vault's contents.

Allow

stateDiagram-v2

```
[*] --> QUEUED
QUEUED --> RUNNING: first frame claimed
QUEUED --> PAUSED: pause before claim
RUNNING --> PAUSING: pause requested
PAUSING --> PAUSED: active work settles
PAUSED --> QUEUED: resume
RUNNING --> FINALIZING: all render frames complete
FINALIZING --> COMPLETED: artifacts committed
RUNNING --> FAILED: unrecoverable frame failure
QUEUED --> CANCELING: cancel with active work
RUNNING --> CANCELING: cancel requested
CANCELING --> CANCELED: active work settles
FAILED --> QUEUED: requeue
```

The generic `CanTransition` table documents lifecycle transitions, while command-oriented methods encode richer semantics. `CommandJob("pause")`, for example, is idempotent when a job is already `PAUSED` or `PAUSING`. A queued job can become paused immediately, but a running job becomes `PAUSING` because a worker may still own a lease. Cancellation first marks unclaimed frames canceled, requests cancellation of active frames, and chooses `CANCELING` or `CANCELED` based on remaining leased work.

Frame tasks have their own statuses: `QUEUED`, `CLAIMED`, `RENDERING`, `COMPLETE`, `RETRYING`, `FAILED`, and `CANCELED`. Separating the aggregate job state from individual frame states allows partial progress, targeted retry, and graceful cancellation. The job's `completed_frames` is a persisted projection recomputed from task rows during transitions; it is not a counter trusted from a worker.

Intermediate states such as `PAUSING`, `CANCELING`, and `FINALIZING` are essential. They make asynchronous work visible instead of pretending a request has instantaneous effects. The cost is more transition code and more tests, but the result is explainable after crashes and races.

Immutable scene versions and frozen job sources

A scene is a stable identity with a `latest_version` pointer. A scene version is an immutable snapshot with its own ID, monotonically increasing version number, original filename, managed storage path, checksum, size, extracted metadata, assets, missing assets, review, and creation time.

`AddSceneVersion` requires `version.Version == latest + 1`, inserts the new version and its assets, advances `scenes.latest_version`, emits `scene.version_created`, and commits.

Existing rows are not overwritten with new scene bytes. Reviews attach to a particular version, so approving version 3 says nothing accidental about version 4.

Display Mermaid diagrams in this vault?

Only allow if you trust this vault's contents.

Allow

erDiagram

```
SCENE ||--o{ SCENE_VERSION : contains
SCENE_VERSION ||--o{ ASSET : references
SCENE_VERSION ||--|| SCENE_REVIEW : has
RENDER_JOB ||--|| JOB_SOURCE : freezes
JOB_SOURCE }o---|| SCENE_VERSION : uses
RENDER_JOB ||--o{ FRAME_TASK : expands_to
```

This immutability solves reproducibility. At job creation, `job_sources.scene_version_id` is written once. Workers later join through that row to receive the exact scene path, even if artists upload and approve newer versions while rendering continues. `job_configs.resolved` similarly freezes complete settings derived from scene metadata and user overrides.

The alternative, resolving `scenes.latest_version` at claim time, would make one job capable of rendering different frames from different scene revisions. It would also make incident analysis nearly impossible. Freezing inputs uses additional rows and storage, but rendering is expensive enough that reproducibility is worth the cost.

Assets use content checksums and an `asset_blobs` table, allowing immutable versions to share identical bytes. Missing assets are modeled explicitly rather than silently omitted. This preserves the distinction between “the version has no asset” and “the version refers to an asset FrameShift could not package.”

Job sources and leases

Workers pull work through the API; they do not read the database or consume an authoritative Go channel. `ClaimFrame` opens a transaction, recovers eligible expired claims, checks worker registration, heartbeat, slot capacity, desired state, and device compatibility, then selects a queued frame according to workspace fairness, priority, age, and frame number.

The claim uses a conditional update:

```
UPDATE frame_tasks
SET status = ?, worker_id = ?, lease_expires_at = ?, attempt = ?
WHERE id = ? AND status = ?
```

Success requires exactly one affected row. This compare-and-set shape prevents two concurrent workers from both becoming owner of the same queued task. The same transaction

creates a render attempt, initializes metrics and progress, updates the parent job, advances workspace scheduler state, and emits `frame.claimed`.

The returned lease contains the frozen scene path, deterministic output path, resolved settings, attempt number, and expiration. A worker periodically renews by presenting its task and worker IDs. Renewal succeeds only if ownership, frame status, job status, and cancellation conditions still agree. Completion and failure use equivalent ownership checks. A late worker cannot commit a result after its lease expired and was reassigned.

Expired leases are expected, not exceptional. `RecoverStalledWork` marks the attempt failed, clears ownership, resets progress, returns the frame to `QUEUED` or completes cancellation, adjusts the job, and records `frame.recovered`. This is the foundation of crash recovery: responsibility is durable and time-bounded.

Authorization belongs beside authoritative queries

Authentication and authorization are different boundaries. API middleware hashes the session cookie, loads the user from a persisted session, and places that user in request context. Worker endpoints use a separate bearer token checked with constant-time comparison. System administration is another explicit check.

After authentication, handlers enforce access and capability requirements. Workspace roles are `OWNER`, `ADMIN`, `ARTIST`, `OPERATOR`, and `VIEWER`. Capabilities describe actions rather than forcing every handler to interpret role names:

- `project.create`
- `scene.write`
- `scene.review`
- `render.create`
- `job.control`

Owners and admins have all capabilities. Artists can create projects, write scenes, render, and control jobs. Operators can render and control jobs. Viewers receive no mutation capability. Review permission is intentionally distinct and can be handled by the review policy.

The API calls methods such as `RequireProjectCapability` before accepting intent, and sensitive store methods such as frame commands check capability again. Queries derive membership through `project_workspaces` and `workspace_members`; job authorization first resolves the job's project. This is stronger than trusting a `workspaceId` supplied by the browser.

Read behavior also avoids information leaks. `requireProjectAccess` returns a not-found response when the user cannot access a project, while forbidden responses explain that an

authenticated role lacks a mutation capability. SSE event queries apply the same ownership graph so one user does not receive another workspace's activity.

Capabilities are easier to audit than scattered role comparisons, but the current mapping is code-defined. A future configurable policy system would add flexibility at the cost of migration, caching, and policy-debugging complexity. For the current product, a small explicit map is predictable.

Durable events and SSE

FrameShift stores events in an append-only `events` table with an integer ID, type, JSON payload, and creation time. State-changing transactions call `insertEvent` before commit. SSE then polls committed events and streams them to authenticated clients.

Display Mermaid diagrams in this vault?

Only allow if you trust this vault's contents.

Allow

`sequenceDiagram`

```
participant UI as Browser
participant API as Go API
participant DB as Database
UI->>API: POST job intent
API->>DB: BEGIN
API->>DB: insert job, source, tasks
API->>DB: insert job.queued event
API->>DB: COMMIT
API-->>UI: 201 Created
UI->>API: GET events/stream + Last-Event-ID
API->>DB: events visible to user after ID
DB-->>API: committed rows
API-->>UI: id / event / data
```

The stream accepts `Last-Event-ID`, reads up to 100 later events in order, emits SSE records, sends keepalives when idle, and repeats once per second. If the network disconnects, the browser can resume from a durable cursor. Events are filtered by payload `userId`, accessible `projectId`, or accessible `jobId`, using SQLite JSON extraction or PostgreSQL JSONB extraction.

SSE is a projection, not a command bus. Clients should react to an event by updating or refetching relevant data, but the event does not outrank current API state. Delivery is effectively at least once across reconnects, so handlers should use event IDs or naturally idempotent refreshes. There is no promise that a connected browser saw an event exactly once.

Why persist events instead of broadcasting directly? Atomicity and replay. A process crash after commit but before an in-memory broadcast would otherwise create an invisible transition. Conversely, broadcasting before commit could announce state that later rolls back.

Why Go channels are not authoritative

Go channels are excellent for coordinating goroutines within one running process. They provide backpressure, ownership transfer, and cancellation patterns. They are not durable: contents disappear on restart, cannot participate in a SQL transaction, and are not naturally shared across multiple server replicas.

FrameShift's long-running loops therefore poll durable claim tables rather than treating a channel as a queue. Finalization, email, publication, and frame execution all have persisted status, attempts, scheduling or lease information. A channel could later be added as a wake-up optimization after commit, reducing poll latency, but losing that signal must not lose the work. The database remains the fallback discovery mechanism.

This distinction also applies to mutexes and React state. `uploadLocks` can serialize chunks efficiently inside one server process, but upload offset and status are persisted. A mutex cannot protect against a second process or a restart. In-memory state may improve coordination; authoritative constraints require durable rows, conditional updates, and database uniqueness.

Managed filesystem paths

The server defaults `FRAMESHIFT_DATA_DIR` to `./var`. It stores SQLite at `var/frameshift.db` and resolves an absolute storage root at `var/storage`. Database paths for managed local files are generally relative, slash-normalized paths beneath that root.

Representative locations are:

```
var/  
  frameshift.db  
  storage/  
    uploads/<upload-id>.part  
    projects/<project-id>/scenes/<scene-id>/versions/<n>/scene.blend  
    projects/<project-id>/scenes/<scene-id>/versions/<n>/thumbnail.png  
    projects/<project-id>/jobs/<job-id>/frames/frame_000001.png  
    projects/<project-id>/jobs/<job-id>/artifacts/final.mp4
```

Relative paths make the data directory relocatable and avoid exposing host paths through JSON; several domain fields use `json:"-"`. Code reconstructs local paths with `filepath.Join(storageRoot, filepath.FromSlash(relative))`. Generated identifiers are validated before constructing scene directories, filenames are sanitized, and file-serving paths

reject absolute or non-local values. Object-backed artifacts use an explicit `object://` form instead of pretending they are local files.

Scene ingestion writes to a temporary file with restrictive permissions, hashes while copying through a size limit, validates the Blender header, and renames to `scene.blend`. Archive ingestion similarly normalizes entries and rejects unsafe traversal. Blender inspection runs without enabling scripts from uploaded scenes. These controls are part of authority: untrusted filenames and embedded code must not be allowed to redefine where or how the server writes.

SQLite, PostgreSQL, and the dialect adapter

Development defaults to SQLite through `modernc.org/sqlite`, with foreign keys enabled and a busy timeout. `FRAMESHIFT_DATABASE_URL` switches to PostgreSQL through `pgx`. The store wraps `sql.DB` and `sql.Tx` in small `database` and `transaction` types carrying a dialect.

Application SQL uses `?` placeholders. For PostgreSQL, `bindQuery` rewrites placeholders outside quoted strings to `$1`, `$2`, and so on. Schema conversion maps SQLite's autoincrement primary key, `INTEGER`, and `DATETIME` forms to PostgreSQL-compatible types. A few semantic differences remain explicit, such as SQLite `json_extract` versus PostgreSQL `payload::jsonb->>` in event authorization.

This adapter has useful properties: SQL stays visible, transaction behavior is direct, and the project avoids an ORM model that could obscure locking and affected-row checks. It is also intentionally limited. It is not a general SQL parser, and textual schema conversion cannot erase every difference in concurrency, typing, JSON, backup tooling, or query planning. New queries must be exercised against both backends when they depend on dialect behavior.

`internal/store/postgres_test.go` exists for that reason.

SQLite is convenient for a single-machine installation and managed local storage. PostgreSQL supports stronger multi-process deployment and external database operations. Supporting both means choosing the common subset deliberately and isolating exceptions rather than assuming identical engines.

Migrations and compatibility

`Store.Open` and `OpenDatabase` run migration before returning a usable store. The schema uses `CREATE TABLE IF NOT EXISTS` and `CREATE INDEX IF NOT EXISTS`, then applies targeted compatibility backfills such as legacy workspaces, scene reviews, storage lifecycle defaults, and job scheduling rows. Backfills use conflict-tolerant inserts so reopening is idempotent.

This approach is simple and appropriate while schema evolution remains additive and controlled. It gives a new installation the same schema path as an upgraded installation. It also has limits: `IF NOT EXISTS` does not prove that an existing table has the expected columns or

constraints, and there is no numbered migration ledger shown here. Destructive changes, column rewrites, or long production migrations would benefit from ordered migration versions, checksums, explicit locking, and rollback or roll-forward procedures.

The authority principle applies to schema as well: the executable must not serve requests against a partially understood database. Migration failure closes the connection and prevents startup. Backup and restore workflows differ by backend (`VACUUM INTO` for SQLite and `pg_dump / pg_restore` paths for PostgreSQL), but both preserve database state together with managed storage manifests.

Idempotency and crash consistency

Distributed render work should assume retries. HTTP clients retry after timeouts, workers reconnect, processes stop between side effects, and an operator may repeat a command.

FrameShift uses several complementary techniques:

1. Stable identities and unique constraints prevent duplicate logical rows, such as one frame task per `(job_id, frame_number)` and one attempt per `(task_id, attempt)`.
2. Conditional updates require the expected current status and owner.
3. Commands define idempotent terminal behavior where useful. Repeating pause during `PAUSING` or cancel during `CANCELING` returns the current state.
4. Upserts make heartbeats, progress, controls, scheduler state, and migration backfills repeatable.
5. Leases convert worker death into recoverable queued work.
6. Outbox rows persist external delivery attempts independently of transient process memory.
7. Checksums and sizes validate that bytes correspond to reported results.
8. Events commit atomically with the state they describe.

SQL cannot atomically commit a filesystem rename or object-store upload. FrameShift therefore aims for crash consistency through ordering and compensation. During scene ingestion, bytes are written and validated under a temporary name, then atomically renamed within the target filesystem. Only afterward can the store transaction register the version. If registration fails, API/service cleanup can remove the uncommitted version directory. This ordering prefers an orphaned file that can be cleaned over a committed row pointing to incomplete bytes.

Frame rendering follows a related pattern. Blender writes through a temporary render directory, output is validated and moved to its deterministic managed path, and the worker reports checksum and dimensions. The store accepts completion only from the current lease owner. A crash after file publication but before database completion can leave an unreferenced or replaceable file; it does not falsely mark the task complete. A retried attempt writes the deterministic result again and must still win the database lease check.

Outboxes address the opposite ordering problem for irreversible external effects. Creating a notification and its email delivery row in one transaction guarantees that committed intent survives a server restart. A background loop claims and sends it, then marks completion or schedules another attempt. Exactly-once email or webhook delivery is generally impossible without cooperation from the receiver: a crash after remote acceptance but before local completion can cause a duplicate. Receivers should accept idempotency keys such as request IDs, and messages should be designed to tolerate duplicate delivery.

Display Mermaid diagrams in this vault?

Only allow if you trust this vault's contents.

Allow

flowchart TD

```
Intent[Validated intent] --> Tx[Begin SQL transaction]
Tx --> Guard[Check state, owner, capability]
Guard --> Rows[Write state and durable work rows]
Rows --> Event[Append event]
Event --> Commit{Commit succeeds?}
Commit -->|no| Rollback[No authoritative change]
Commit -->|yes| Discover[Workers, outboxes, SSE discover work]
Discover --> Lease[Lease / attempt / idempotency guard]
Lease --> SideEffect[Filesystem or external side effect]
SideEffect --> Report[Commit validated result]
```

The practical contract is not “no duplicate execution.” It is “duplicate or interrupted execution cannot silently create duplicate authoritative completion.” That is the more achievable and useful guarantee.

Tradeoffs and design pressure

Database authority adds latency and code compared with pushing structs through channels. Polling SSE and outboxes is less immediate than an in-memory broadcast. Immutable versions consume storage. Fine-grained state machines require tests for transitions and recovery. Supporting two SQL dialects constrains query design.

Those costs purchase restartability, auditability, horizontal evolution, and reproducibility. A user can refresh the UI, a worker can vanish, and the server can restart without asking memory what happened. Every job can identify its source version and resolved settings. Every accepted result has a worker, attempt, lease history, checksum, and durable transition.

There are also natural future improvements: versioned migrations, stronger PostgreSQL claim locking for high contention, event retention and consumer cursors, explicit idempotency keys on command endpoints, reconciliation jobs for orphaned files, and a transactional-outbox

dispatcher with wake-up notifications. Each should preserve the existing rule: optimizations may accelerate discovery, but they must not become a second source of truth.

A review checklist

When adding a feature to FrameShift, ask these questions:

- Which store method owns the decision?
- What current state, membership, capability, or lease must be checked?
- Which rows must commit together to avoid a half-state?
- Is the source input frozen, checksummed, and reproducible?
- Can the operation be retried after an ambiguous timeout?
- What happens if the process stops immediately before or after each external side effect?
- Is durable work discoverable without an in-memory signal?
- Does the event commit with the state, and is it filtered to authorized users?
- Are managed paths relative, validated, and contained beneath the storage root?
- Does the SQL behave on both SQLite and PostgreSQL?
- Is there a recovery or reconciliation path for expired leases, pending outboxes, and orphaned bytes?

An authoritative render architecture is not created by declaring one service “the backend.” It emerges from precise ownership of decisions. In FrameShift, Go store transactions are that ownership boundary. Workers execute leases, the frontend expresses intent, SSE communicates committed changes, and storage preserves bytes. Because none of those projections can overrule the database, the system remains understandable when the happy path ends.

With authority located in transactions and managed storage, the first hostile boundary is the scene-ingestion path.

Part 4 — Treating Blender Files as Untrusted Input

Uploading a Blender scene is not an ordinary file-copy operation. A `.blend` file is a complex, versioned binary document interpreted by a large native application. A ZIP scene package adds attacker-controlled names, compression, entry metadata, and potentially thousands of assets. Even a legitimate production scene can consume extreme CPU, memory, disk, or render time. FrameShift therefore treats every uploaded byte as untrusted until a bounded ingestion pipeline has identified it, inspected it with constrained Blender settings, and recorded the resulting scene version in the authoritative database.

This chapter describes that pipeline as implemented by the Go backend. The important architectural rule is that the browser supplies intent and bytes, but does not decide whether an

upload is complete, valid, unique, within quota, or eligible to become a scene version. SQLite or PostgreSQL and managed storage remain the system of record.

Threat model and trust boundaries

The pipeline assumes that an authenticated user may accidentally or deliberately upload hostile input. Relevant threats include:

- a renamed non-Blender file intended to reach Blender or another parser;
- an embedded Blender script or driver that expects automatic execution;
- shell metacharacters in a filename or generated path;
- oversized request bodies and clients that understate their size;
- ZIP traversal, absolute paths, backslash ambiguity, NUL bytes, symlinks, file-count attacks, and compression bombs;
- packages with no clear main scene, or with multiple `.blend` files chosen through ordering tricks;
- duplicated content used to consume logical or physical storage;
- missing external files that make a scene non-reproducible;
- interrupted uploads, process restarts, and two clients racing at the same offset;
- replacement uploads that mutate historical work or invalidate jobs already bound to a version;
- Blender parser vulnerabilities and resource exhaustion during inspection.

Authentication and authorization are necessary but do not make content trusted. FrameShift checks the caller's `scene.write` capability and verifies that a replacement scene belongs to the selected project. Those checks answer *who may attempt an upload*. The ingestion checks answer *whether these bytes may be retained and published*.

End-to-end ingestion sequence

The resumable upload API and the scene service form two layers. The API stages bytes and owns durable upload state. The scene service validates and materializes a version. The store publishes that version transactionally only after all earlier work succeeds.

Display Mermaid diagrams in this vault?

Only allow if you trust this vault's contents.

Allow

```
sequenceDiagram
    actor Client
    participant API as Go API
    participant DB as Authoritative store
    participant Stage as Upload staging
    participant Scenes as Scene service
```

```

participant Blender
participant Managed as Managed scene storage

Client->>API: POST upload metadata (size, optional SHA-256)
API->>DB: Check capability, ownership, and quota headroom
API->>Stage: Create uploads/upl_*.part (0600)
API->>DB: Insert UPLOADING session, offset 0, expiry
loop Chunks of at most 4 MiB
    Client->>API: PATCH with Upload-Offset
    API->>DB: Read authoritative offset
    API->>Stage: WriteAt(offset), fsync, close
    API->>DB: Compare-and-swap offset
    API-->>Client: 204 and next Upload-Offset
end
Client->>API: POST complete
API->>DB: UPLOADING to FINALIZING
API->>Stage: Re-read, verify size and optional SHA-256
API->>Scenes: Ingest generated IDs and staged reader
Scenes->>Managed: Bounded temporary copy or safe ZIP extraction
Scenes->>Scenes: Verify container and package policy
Scenes->>Blender: Inspect with --disable-autoexec
Blender-->>Scenes: Metadata marker and rendered thumbnail
Scenes-->>API: SceneVersion, assets, checksums
API->>DB: Recheck expanded quota and project duplicate
API->>DB: Transactionally insert immutable version
API->>Stage: Delete session and .part file
API-->>Client: Created scene or version

```

There are two meanings of promotion in this flow. First, files are renamed from temporary names to generated managed paths only after their stream-level checks pass. Second, and more importantly, a scene is not authoritative until the database transaction inserts the scene or next version. If inspection, quota rechecking, duplicate detection, or database publication fails, the scene service removes the generated version directory.

Recognizing a Blender container

An extension is only an admission hint. The upload API accepts filenames ending in `.blend` or `.zip`, case-insensitively, but ingestion examines the bytes before invoking Blender. For a direct scene, `verifyBlendHeader` reads seven bytes and accepts one of three known container signatures:

```

uncompressed := string(header) == "BLENDER"
gzipCompressed := header[0] == 0x1f && header[1] == 0x8b

```

```
zstdCompressed := header[0] == 0x28 && header[1] == 0xb5 &&  
    header[2] == 0x2f && header[3] == 0xfd
```

The literal `BLENDER` header identifies an uncompressed Blender file. The two-byte `1f 8b` signature identifies gzip, and `28 b5 2f fd` identifies a Zstandard frame. Supporting compressed Blender containers matters because Blender can produce or consume them; rejecting everything that does not begin with `BLENDER` would incorrectly reject valid scenes.

This is deliberately a cheap identification check, not proof that the complete file is semantically valid or safe. A short attacker-controlled payload can reproduce a magic number. Full structural interpretation is delegated to Blender during controlled inspection, and even successful parsing is not a guarantee that Blender has no parser vulnerability. The header gate reduces accidental parser exposure and gives clients a useful validation error; it is not a sandbox.

Bounded streaming before parsing

The backend never trusts a declared content length by itself. Upload creation rejects non-positive sizes and sizes above `FRAMESHIFT_MAX_UPLOAD_BYTES`, which defaults to 10 GiB. Every `PATCH` body is limited to 4 MiB, and a chunk cannot cross the session's declared total. Completion then re-reads the assembled staging file and requires its actual byte count to equal the expected size.

The scene service applies its own bound again. A direct `.blend` is copied through `io.LimitReader(source, maxBytes+1)`. Reading one byte beyond the limit makes truncation distinguishable from an exactly full upload. The same compressed-byte bound applies when a ZIP is first copied to a temporary archive. Layered bounds are intentional: an internal caller cannot bypass ingestion limits merely because it did not arrive through the HTTP route.

Temporary direct uploads are created in the generated version directory, changed to mode `0600`, hashed while streaming, closed, header-checked, and renamed to `scene.blend`. Closing before inspection ensures Blender sees a stable file. A failed copy, close, header check, rename, or inspection triggers cleanup of the version directory.

Generated paths, not uploaded paths

FrameShift generates project, scene, scene-version, and upload identifiers. Scene storage methods accept only identifiers matching prefixes such as `prj_`, `scn_`, and `scv_` followed by 24 lowercase hexadecimal characters. The resulting layout is predictable to the backend but not controlled by the uploaded filename:

```
storage/  
  uploads/upl_<generated>.part  
  projects/prj_<generated>/scenes/scn_<generated>/versions/1/
```

```
scene.blend
thumbnail.png
```

For a direct upload, the original filename is display metadata only. The backend takes its base name and replaces characters outside `[a-zA-Z0-9._-]` with underscores. It does not join that name into the managed scene path. Thus a name such as `../unsafe scene.blend` may be retained as `unsafe_scene.blend` for display while the bytes live at generated `scene.blend`.

ZIP entry paths are different because Blender needs the package's relative directory structure to resolve textures and linked libraries. Those paths are retained only after archive-specific normalization and containment checks described below.

Blender inspection without automatic scripts

FrameShift invokes Blender to obtain semantic metadata and a real preview. Every invocation uses `exec.CommandContext` with a fixed argument array:

```
command := exec.CommandContext(commandContext, blenderPath,
    "--background", "--factory-startup", "--disable-autoexec", scenePath,
    "--python-expr", inspectionScript,
    "--", thumbnailPath,
)
```

There is no shell and no interpolated command string. A filename containing spaces, quotes, semicolons, `$()`, or leading dashes remains one argument; it cannot become shell syntax. The executable path is server configuration, while scene and thumbnail paths are generated by the backend. `CommandContext` also gives the process a cancellation path, and inspection wraps the request context in a 45-second timeout by default.

The security-critical flag is `--disable-autoexec`. It tells Blender not to automatically execute Python stored in the file or referenced by auto-run mechanisms when opening the scene. `--factory-startup` avoids loading a user's normal startup file and most user preference state. `--background` avoids an interactive UI and makes the process suitable for controlled server execution.

The `--python-expr` script is different from scene-controlled auto-execution: it is static FrameShift code passed as a distinct argument. It reads known Blender data structures, emits one compact JSON record prefixed with `FRAMESHIFT_METADATA=`, and receives the thumbnail destination only after Blender's `--` separator. The Go process scans combined output for that exact marker and JSON-decodes it. Arbitrary Blender log lines are not interpreted as metadata.

`--disable-autoexec` is a strong and necessary control, but it is not process isolation. Blender still parses attacker-controlled native data, allocates memory, resolves paths, loads some

external resources, and renders a preview. A vulnerability in Blender or a dependency can execute within the Blender process. Production deployments should therefore combine application controls with operating-system isolation, least-privilege accounts, resource limits, outbound-network policy, and prompt Blender security updates.

Inspection metadata and real thumbnails

Inspection produces facts used by later UI and scheduling decisions rather than trusting client claims. The current metadata includes:

- Blender version and render engine;
- frame start, frame end, and effective FPS;
- X/Y resolution and percentage scale;
- cameras, active camera, scenes, and view layers;
- output format;
- Cycles sample count and denoising state, with controlled defaults for other engines;
- film transparency;
- unresolved image, movie clip, sound, font, and linked-library paths;
- an explicit `autoExecDisabled` fact, set by both the script and Go code.

The thumbnail is not a filename-derived icon or browser placeholder. The inspection script temporarily selects `BLENDER_WORKBENCH`, sets a 480 by 270 PNG target, disables transparency, moves to the first scene frame, and calls Blender's real render operator. The API later serves `thumbnail.png` from the selected immutable version directory.

Thumbnail generation is intentionally separated from production render settings. It provides a fast visual identity for review without running the scene's selected production engine. At present, the Python script catches thumbnail errors, prints `FRAMESHIFT_THUMBNAIL_ERROR`, and can still return metadata. That favors metadata availability, but it means a successfully ingested version may have no thumbnail and the thumbnail route may return not found. A stricter installation could make a valid PNG mandatory, but doing so would reject scenes that are inspectable yet not previewable in Workbench.

Atomic promotion and failure cleanup

Direct scene bytes first land in `.upload-*` beside their final destination. ZIP entries first land in `.extract-*` files. Permissions and checksums are established before `os.Rename` promotes each file. On a single managed filesystem, rename avoids exposing a partially copied final file to another process. Content-addressed asset blobs are similarly renamed from an extracted temporary file only when that checksum does not already exist.

This guarantee has precise limits. The archive package is assembled entry by entry; the entire package directory is not swapped into place with one rename. It remains unpublished to normal application reads because no database version points at it until ingestion succeeds. Cleanup removes the candidate version directory on an error. Database insertion for a replacement then verifies that its version number is exactly `latest + 1`, inserts metadata, review state, assets, and events in one transaction, and updates `latest_version` in that transaction.

Filesystem rename and database commit are not one cross-system transaction. A crash can leave unreferenced managed files after promotion but before commit. Conversely, the implementation publishes the database only after files exist, avoiding records that intentionally point to incomplete content. Operational reconciliation should find generated version directories with no database row and remove them after a grace period. Managed storage and the database should be backed up consistently.

Defending ZIP extraction

A ZIP package is accepted because many Blender scenes depend on adjacent textures, fonts, caches, linked libraries, or models. It also greatly expands the attack surface. FrameShift performs a complete metadata pass before extracting any entry.

First, the archive must open successfully and contain between 1 and 10,000 entries. The sum of declared uncompressed sizes must not exceed four times the normal upload limit. The sum of compressed sizes must be nonzero, and declared expansion must not exceed a 100:1 ratio. These checks bound common ZIP bombs before decompression begins.

Second, every name passes `safeArchivePath`. Names containing NUL, backslash, or an initial slash are rejected. POSIX-style `path.Clean` then rejects `.`, `..`, paths beginning with `../`, and absolute paths. Using `path`, rather than the host-specific `filepath`, gives ZIP names one canonical slash grammar. Rejecting backslashes avoids a package that is harmless on Unix but traverses when interpreted with Windows separators. Symlink-mode entries are rejected before extraction, so an attacker cannot arrange for a later file write to follow a link outside the package.

Third, extraction remains bounded even after trusting neither the central directory nor the initial arithmetic. Each entry is copied through a reader limited to its declared uncompressed size plus one byte. If the decompressor produces more than declared, extraction fails. Temporary files are promoted only after complete extraction and close.

The controls are complementary:

- traversal checks constrain *where* bytes may be written;
- symlink rejection prevents indirection around that constraint;
- the file-count limit constrains metadata and filesystem overhead;

- the total expanded-size cap constrains aggregate disk use;
- the ratio cap catches extreme compression;
- per-entry bounded copies defend against dishonest size declarations.

No single ZIP-bomb heuristic is perfect. A package just under every threshold can still be expensive, and 10,000 small files can stress antivirus, backup, and filesystem metadata. Deployments should choose `FRAMESHIFT_MAX_UPLOAD_BYTES` for the host's real disk reserve and may lower archive limits for smaller installations.

Exactly one Blender file

Archive order never decides the main scene. During extraction, FrameShift counts files whose normalized extension is `.blend`, case-insensitively. Zero and two-or-more both fail with “scene package must contain exactly one `.blend` file.” The selected file must also pass the Blender container header check.

This rule is intentionally simple. A manifest-selected entry would add another attacker-controlled parser and ambiguous fallback behavior. Exactly one scene gives the package one deterministic root while still allowing linked assets of other supported types. It also prevents a user interface from inspecting one file while a worker later renders another.

Assets, blobs, hardlinks, and missing references

Every non-directory, non-`.blend` archive entry becomes a present asset. FrameShift streams it through SHA-256 while extracting, records its actual size, and classifies it by extension as a texture, model, font, cache, metadata file, or other asset. The hash determines a generated blob path:

```
blobs/<first-two-hash-characters>/<full-sha256>
```

If the blob does not exist, the temporary asset is atomically renamed there. If it already exists, the duplicate temporary file is removed. The package-relative target is then created as a hardlink to the blob. This preserves the exact relative path Blender expects while physically reusing unchanged content across scene versions. If hardlink creation fails, for example across filesystems or because the filesystem does not support links, FrameShift copies the blob into the package as a compatibility fallback.

Checksums make blob identity content-based rather than filename-based. Asset IDs are derived from the version ID and normalized package path, so two paths with identical bytes remain two manifest entries while sharing a blob. This is physical deduplication, not authorization deduplication: access is still through a scene version and project membership, never by accepting an arbitrary checksum from a client.

Blender inspection also reports referenced paths that do not resolve on disk. FrameShift normalizes those strings and records deterministic missing assets with `missing=true` and `external=true`. Missing records have no checksum, size, or blob. Keeping them is important: silently dropping an unresolved texture or linked library would make a scene appear complete when it is not. Review and comparison can expose the problem before production rendering.

Hardlinks carry a mutability warning. Modifying a package hardlink in place would modify the shared blob and every linked version. FrameShift treats version storage as immutable and does not offer an in-place asset editor. Maintenance tools must preserve that rule: replacements should create new blobs and links, never open an existing package asset with truncation or write access.

Checksums and duplicate scenes

For a direct upload, SHA-256 is computed during the bounded copy. For a ZIP, the scene version checksum is the checksum of the single extracted `.blend`, while each asset has its own checksum. The client may provide an optional whole-upload SHA-256 when creating a resumable session. Completion computes the checksum of the assembled `.part` file and rejects a mismatch before ingestion.

These hashes answer different questions. The optional session checksum detects corruption or a changed upload stream. The scene checksum identifies the Blender content selected from either a direct file or package. Asset checksums provide content-addressed storage and version comparison. Because a ZIP checksum would change with entry order or compression settings, using the extracted `.blend` checksum makes project duplicate detection stable across differently packed archives.

Before database publication, the API asks whether the project already has that scene checksum. A duplicate candidate is removed and the request returns a conflict. This avoids duplicate scene content, but it also means two packages with the same `.blend` and different bundled assets are considered duplicate scenes. That is a deliberate current policy and may be too coarse if package assets are semantically part of scene identity. A future package identity could hash a canonical manifest of the scene checksum plus sorted `(path, asset checksum)` pairs.

Immutable replacements and comparison

A replacement upload carries an existing `sceneId`. The API verifies project ownership, reads the latest version, and asks ingestion to build `latest + 1` under a new generated version ID. `AddSceneVersion` checks that sequence again inside its transaction. It never overwrites the prior directory or row.

Immutability keeps existing jobs reproducible: a render bound to version 2 does not silently start reading version 3. It also makes review history meaningful and prevents an approved version from changing after approval. A failed replacement removes only its candidate version directory and leaves the scene's latest pointer unchanged.

Comparison is computed from stored facts. Metadata values are converted to maps and reported when before and after differ. Assets are keyed by package-relative path: a new path is added, a missing old path is removed, and a path whose checksum changed is changed. Missing references participate as asset records, so resolving or introducing an external dependency can be visible. This is a structural comparison, not a visual or semantic Blender diff; renamed assets appear as remove plus add, and object-level mesh changes are represented only indirectly through the scene checksum and inspected metadata.

Resumable 4 MiB upload protocol

Creating an upload stores expected size, optional lowercase SHA-256, owner, project, optional replacement scene, generated staging path, status, expiry, and offset zero. Sessions expire after 24 hours. The staging file is created exclusively with mode `0600`, preventing accidental reuse of an existing path.

Each PATCH supplies `Upload-Offset` and a body no larger than 4 MiB (`4 <= 20` bytes). The server locks the upload ID within the process, reads the durable session, and requires the supplied offset to equal the database offset. A mismatch returns `409` and the authoritative offset in the response header. Valid bytes are written with `WriteAt`, synchronized to disk, and closed before the offset advances.

The database update is itself a compare-and-swap:

```
UPDATE upload_sessions
SET upload_offset = ?
WHERE id = ? AND user_id = ? AND status = 'UPLOADING'
AND upload_offset = ? AND expires_at > ?
```

That predicate prevents a stale writer from advancing state after another request or after expiry. The protocol is resumable by asking `GET /api/v1/uploads/{id}` for the current offset and continuing there. Cancellation deletes both the row and staging file. Completion requires `offset == expected_size`, changes `UPLOADING` to `FINALIZING`, verifies the assembled file, and only then enters scene ingestion.

Restart recovery and interrupted finalization

An ordinary server restart does not erase upload progress. Session metadata and offset live in the database, while bytes live under managed `uploads/`. The in-memory mutex map is only a

same-process concurrency optimization; after restart, a new lock is created on demand, and the database offset predicate remains the authoritative race check. A client can read the stored session and resume from the persisted offset.

There is a durability ordering tradeoff. A chunk is fsynced before its database offset advances. A crash between those operations may leave bytes beyond the recorded offset. Resuming writes those bytes again at the same position, which is safe because the database tells the client where to restart. Advancing the offset first would be worse because the database could claim bytes that never reached disk.

The current implementation has a gap for crashes after transition to `FINALIZING`. Unlike render and publishing work, upload finalization has no lease expiry or startup reset, and `PATCH` accepts only `UPLOADING`. A process death can therefore strand a session in `FINALIZING`, possibly with a candidate directory. Robust recovery should, on startup or a periodic sweep, reconcile stale finalizing rows: check whether a scene/version transaction committed, remove unpublished candidate paths, verify the staging file, and either retry idempotently or return the session to `UPLOADING`. Expired `UPLOADING` rows and orphan `.part` files likewise need a garbage-collection sweep; checking expiry during `PATCH` alone does not reclaim disk.

Quota checks and reservation

FrameShift checks workspace capacity when a session is created using the declared compressed/direct upload size. It checks again after ingestion using the scene bytes plus every present asset's logical size. The second check matters for ZIP expansion. Missing assets do not consume quota. On failure, the candidate version is removed and the upload can return to `UPLOADING`.

Current `WorkspaceCanStore` calls are preflight checks, not a strict reservation. The `upload_sessions` row does not store reserved bytes in quota accounting, and the check and session insert are not one workspace-locked transaction. Concurrent uploads can each observe enough headroom and collectively overcommit. The completion-time check narrows the risk, but two completions can also race before their scene transactions become visible. In addition, quota accounting charges each version's asset references even when hardlinks reduce physical usage; this conservative logical accounting is predictable but does not equal `du` output.

A strict reservation design should add `reserved_bytes` to active upload sessions and update workspace usage and reservation in one transaction. Session creation would atomically require:

```
committed logical bytes + active reservations + requested reservation <=
quota
```

Cancellation, expiry, and terminal failure would release the reservation. Completion would replace the initial reservation with the actual expanded logical size atomically with scene publication, rejecting or extending it before commit if ZIP expansion needs more. Recovery must reclaim stale reservations alongside stale sessions. Until that mechanism exists, documentation and UI should describe current behavior as quota admission and rechecking, not guaranteed reservation.

macOS security tradeoffs

FrameShift is local-first and commonly runs Blender from `/Applications/Blender.app/Contents/MacOS/Blender` under a LaunchAgent. This is convenient and preserves Metal device access, but macOS process boundaries deserve explicit treatment.

File modes such as `0600`, `0640`, and directory mode `0750` reduce accidental access by other local users, yet a Blender child normally runs as the same account as the FrameShift server or worker. It can therefore access anything that account can access unless macOS privacy controls, a sandbox, or a dedicated service account says otherwise. `--factory-startup` and `--disable-autoexec` do not prevent a native exploit from reading the user's files or opening network connections.

The strongest local deployment uses a dedicated non-admin account with access only to managed storage and the Blender executable. Keep managed storage outside personal Documents/Desktop locations to avoid broad privacy grants, and do not grant Full Disk Access. Restrict outbound networking for the inspection process where feasible. LaunchAgent environment variables and logs should not contain upload content or secrets. Blender and macOS should be patched promptly, and third-party Blender add-ons should not be installed into the service account's profile.

APFS supports same-volume atomic rename and hardlinks, which match FrameShift's promotion and dedup model. Moving `storage/blobs` or version directories onto separate volumes can make hardlinks fail; FrameShift falls back to copying assets, increasing disk use. Atomic rename also depends on temporary and final paths sharing a filesystem, which the current sibling-temp layout provides. Backups, antivirus, Spotlight, and Time Machine can observe files during candidate construction even though the application has not published them, so highly sensitive installations should exclude staging paths or use encrypted dedicated storage.

Finally, a timeout limits wall-clock inspection but is not a complete CPU, memory, child-process, file-descriptor, or disk-I/O sandbox. macOS lacks the simple container primitives commonly used on Linux servers. Practical options include a dedicated account, `launchd` resource policy where available, a hardened helper service, a lightweight virtual machine, or remote Linux workers with stronger isolation. Each option trades setup complexity, preview latency, filesystem

integration, and Metal acceleration against containment. For a single trusted household workstation, same-user execution may be acceptable; for a multi-user or internet-exposed service, Blender inspection should cross a stronger isolation boundary.

Security invariants to preserve

Future changes to ingestion should preserve a small set of reviewable invariants:

1. Uploaded names never select top-level managed paths.
2. Every byte copy and decompression has a server-side bound.
3. ZIP paths use one canonical grammar and cannot traverse or redirect through symlinks.
4. A package has exactly one verified `.blend` root.
5. Blender is invoked without a shell, with automatic script execution disabled and a deadline.
6. Metadata and thumbnails come from controlled inspection, not client assertions.
7. Checksums are computed by the backend from stored bytes.
8. Candidate files are promoted before, but become authoritative only with, a successful database publication.
9. Replacements append immutable versions and never rewrite historical inputs.
10. Restart, expiry, quota, and cleanup behavior are designed together so staged data cannot become permanent unaccounted storage.

The core lesson is not that a valid signature or `--disable-autoexec` makes a Blender file safe. Security comes from composing independent controls: authenticated intent, generated paths, bounded persistence, archive containment, deterministic scene selection, content hashing, constrained native inspection, immutable publication, and recoverable state. Each control has a narrow job, and the pipeline remains understandable because no single parser, filename, client claim, or frontend state is allowed to decide trust on its own.

Once an immutable scene version is accepted, the platform must turn it into durable work that survives competing workers and failure.

Part 5 — Durable Scheduling: Frames, Leases, Retries, and Recovery

FrameShift treats scheduling as durable state management, not as an in-memory queue. The Go backend and its database are the system of record. Workers ask for work, report progress, and submit results, but they do not decide whether a job is running, failed, canceled, or complete. Those decisions are made by transactional store operations over job, frame, attempt, control, progress, and finalization rows.

This design matters because rendering is long-running and failure-prone. Blender can crash, a worker can lose power, a network request can be duplicated, and the server can restart while

frames are in flight. A durable scheduler must preserve enough information to determine what happened and safely make unfinished work available again. FrameShift does that by decomposing each render job into one persistent task per frame, assigning those tasks with renewable leases, and deriving the job's aggregate state from its frame rows.

Durable Units of Work

Creating a normal render job inserts all scheduling state in one transaction:

- one `render_jobs` row for the user-visible job;
- one `job_sources` row that pins the selected scene version;
- one `job_configs` row containing resolved render settings;
- one `job_scheduling` row containing `priority` and `not_before`;
- one `finalization_tasks` row, initially `WAITING`;
- one `frame_tasks` row for every frame in the requested inclusive range; and
- one durable event describing the queued job.

The frame task identifier is deterministic: the job ID plus a zero-padded frame number. The database also enforces `UNIQUE(job_id, frame_number)`. Together, those properties prevent one logical frame from being represented by multiple task rows. The output path is deterministic as well:

```
projects/{projectID}/jobs/{jobID}/frames/frame_{frameNumber:06d}.png
```

The task row is the current snapshot. It records status, worker ownership, lease expiry, cumulative attempt number, timestamps, output metadata, and the latest error. Related tables retain details that should not be overwritten: `render_attempts` stores each execution attempt, `attempt_metrics` stores resource measurements for that attempt, `frame_progress` stores the latest sub-frame progress, and `logs` stores bounded worker and Blender output.

This separation is intentional. A frame retry resets transient output fields on `frame_tasks`, but it does not reset the task's attempt counter or delete previous `render_attempts`. Operational recovery can therefore distinguish “the same frame tried twice” from “two different frames ran once.”

Calibration jobs use the same durable frame machinery but create only representative start, middle, and end tasks. Their finalization row is `SKIPPED`, and successful calibration frames move the job directly to `COMPLETED`.

Job State Machine

The domain model defines the complete job lifecycle. Some early states are used by broader ingestion and validation flows; scheduler-created render jobs enter directly at `QUEUED` .

Display Mermaid diagrams in this vault?

Only allow if you trust this vault's contents.

Allow

stateDiagram-v2

```
[*] --> DRAFT
DRAFT --> VALIDATING
DRAFT --> CANCELED
VALIDATING --> READY
VALIDATING --> FAILED
VALIDATING --> CANCELING
READY --> QUEUED
READY --> CANCELED
QUEUED --> RUNNING: first frame claimed
QUEUED --> PAUSED: pause before work starts
QUEUED --> CANCELING
RUNNING --> PAUSING: pause requested
PAUSING --> PAUSED: active frame settles
RUNNING --> FINALIZING: all frames complete
RUNNING --> FAILED: terminal frame failure
RUNNING --> CANCELING
PAUSED --> QUEUED: resume
PAUSED --> CANCELING
FINALIZING --> COMPLETED
FINALIZING --> FAILED
FAILED --> QUEUED: requeue
FAILED --> CANCELED
CANCELING --> CANCELED
```

The transition table in the domain package documents allowed generic transitions, while scheduler commands apply stricter command-specific rules. More importantly, the scheduler does not trust a worker to advance job state. Claiming the first frame conditionally changes `QUEUED` to `RUNNING` . Completing or failing a frame reloads an aggregate of all frame rows and updates `completed_frames` and job status in the same transaction as the frame result.

The aggregate divides frame tasks into four categories:

- complete: `COMPLETE` ;
- pending: `QUEUED` , `RETRYING` , `CLAIMED` , or `RENDERING` ;
- failed: `FAILED` ; and
- canceled: `CANCELED` .

When no pending frames remain, failures take precedence, then cancellations, then successful finalization. Thus a job does not enter `FINALIZING` merely because one worker says its last local frame completed. It enters only after the database confirms that every durable frame task is complete and none is failed or canceled.

Frame State Machine

Frames have a smaller lifecycle optimized for independent execution and recovery.

Display Mermaid diagrams in this vault?

Only allow if you trust this vault's contents.

Allow

stateDiagram-v2

```
[*] --> QUEUED
QUEUED --> CLAIMED: conditional claim
CLAIMED --> RENDERING: first renewal
RENDERING --> RENDERING: renew lease
CLAIMED --> COMPLETE: completion before renewal
RENDERING --> COMPLETE: validated result
CLAIMED --> QUEUED: failure below attempt limit
RENDERING --> QUEUED: failure below attempt limit
CLAIMED --> FAILED: attempt limit reached
RENDERING --> FAILED: attempt limit reached
CLAIMED --> QUEUED: expired lease recovered
RENDERING --> QUEUED: expired lease recovered
QUEUED --> CANCELED: frame or job cancel
CLAIMED --> CANCELED: cancel acknowledged or lease expires
RENDERING --> CANCELED: cancel acknowledged or lease expires
FAILED --> QUEUED: individual retry or job requeue
CANCELED --> QUEUED: individual retry or job requeue
COMPLETE --> [*]
```

`RETRYING` is part of the domain and is counted as pending by aggregate and command queries. The current automatic failure path, however, returns retryable work directly to `QUEUED`; progress is labeled `RETRYING`. Code that consumes the model should not assume that the visible progress phase and durable task status have identical names.

Completion records checksum, byte size, dimensions, attempt duration, exit code, startup time, peak memory, peak CPU, and logs transactionally. It also clears the lease. The current task row retains the worker ID after successful completion, which is useful attribution but no longer represents an active lease because only `CLAIMED` and `RENDERING` count as active.

Claiming and Leasing

Workers pull work by calling `ClaimFrame`. A lease is temporary authority to operate on one frame; it is not ownership forever. The returned `FrameLease` includes task and job IDs, frame number, incremented attempt, scene and output paths, resolved settings, and an expiration time.

Claiming runs in a transaction. Before selection, it opportunistically recovers expired active tasks: work from running or queued jobs goes back to `QUEUED`, while work belonging to canceling or canceled jobs becomes `CANCELED`. A separate server recovery loop performs richer bookkeeping every five seconds, as described below.

The scheduler then checks the worker. A registered worker must have a heartbeat newer than 30 seconds, must not be draining, and must have fewer active leases than configured slots. Its device inventory must support the job's resolved settings. CPU or unspecified work is universally compatible; accelerated work requires a matching device type and, when specified, matching device ID. For compatibility with simple/local callers, an unregistered worker is treated as active with one slot and bypasses device filtering.

Candidate selection applies these gates:

```
WHERE frame.status = 'QUEUED'
      AND job.status IN ('QUEUED', 'RUNNING')
      AND (schedule.not_before IS NULL OR schedule.not_before <= now)
```

It examines at most 100 ordered candidates and chooses the first device-compatible one. The order is:

1. the workspace's oldest `last_claim_at`, falling back to job creation time;
2. higher job priority;
3. older job creation time; and
4. lower frame number.

After choosing a candidate, the scheduler does not blindly update it. The write is conditional:

```
UPDATE frame_tasks
SET status = 'CLAIMED', worker_id = ?, lease_expires_at = ?, attempt = ?
WHERE id = ? AND status = 'QUEUED';
```

Exactly one affected row is required. If another claimant changed the task first, this transaction does not acquire it. The claim transaction also creates the `RUNNING` attempt row and metrics row, records initial `CLAIMED` progress, conditionally starts the job, advances the workspace's `last_claim_at`, clears stale frame-control intent, and emits `frame.claimed`. Either that whole set commits or none of it does.

This is an atomic conditional claim, but it is not a reservation made during the candidate `SELECT`. That distinction affects database concurrency and is discussed later.

A Normal Worker Exchange

Display Mermaid diagrams in this vault?

Only allow if you trust this vault's contents.

Allow

```
sequenceDiagram
    participant W as Worker slot
    participant A as Go API
    participant S as Store transaction
    participant DB as Database
    participant B as Blender

    W->>A: claim(worker ID)
    A->>S: ClaimFrame
    S->>DB: recover expired rows; select candidate
    S->>DB: UPDATE ... WHERE status = QUEUED
    S->>DB: insert attempt, metrics, progress, event
    DB-->>W: lease with attempt and expiry
    W->>A: renew(task, worker)
    A->>DB: conditional lease extension
    W->>B: render frame
    loop every 2 seconds while rendering
        W->>A: renew lease
        W->>A: append live logs
        A->>DB: persist ownership, progress, and logs
    end
    B-->>W: validated image and metrics
    W->>A: complete(task, worker, result)
    A->>DB: complete frame and attempt; recompute job
    alt more frames remain
        DB-->>W: job remains RUNNING
    else all frames complete
        A->>DB: job FINALIZING; finalizer QUEUED
    end
end
```

The worker renews once before scene preparation, then every two seconds during rendering. A successful renewal changes `CLAIMED` to `RENDERING` and extends `lease_expires_at`. Every renewal is conditional on the task ID, worker ID, active frame status, an eligible job status, and absence of a frame cancel request:

```
task belongs to worker &&
task is CLAIMED or RENDERING &&
```

```
job is RUNNING or PAUSING &&  
cancel has not been requested
```

If no row matches, the store returns `ErrLeaseConflict`. The worker cancels its render context when a periodic renewal fails, causing Blender execution to stop. Completion and failure reports also require the same worker ownership and active task status, so a late process cannot overwrite a task after its lease has been recovered and reassigned.

FrameShift provides at-least-once execution, not magical exactly-once rendering. A worker may produce an image and crash before committing completion. Once its lease expires, the frame can run again. Deterministic output paths, ownership checks, validation metadata, and transactional state changes make duplicate execution manageable; consumers must not infer exactly-once execution from a single durable completion row.

Attempts and Automatic Retries

The attempt counter belongs to the frame task and increases only when a new claim succeeds. A corresponding immutable attempt identity is formed from task ID and attempt number. Each claim inserts a `RUNNING` attempt and an empty metrics row. Completion changes that attempt to `COMPLETE`; explicit render failure or stale-lease recovery changes it to `FAILED` and records the available diagnostics.

On explicit failure, `FailFrameWithResult` chooses the next state transactionally:

- if the job is canceling or canceled, the frame becomes `CANCELED` and the job becomes `CANCELED`;
- if that frame has a durable cancel request, the frame becomes `CANCELED`;
- if `attempt >= maxAttempts`, the frame becomes `FAILED`; or
- otherwise, the frame returns to `QUEUED` for another claim.

The failed attempt remains in history even when the task is queued again. A later claim increments the task to the next attempt. When a terminal frame failure leaves no pending work, the aggregate job becomes `FAILED`, and the store creates failure notifications and email outbox work in the same transaction.

There is currently no time-based backoff field for frame retries. A retryable failure returns immediately to the eligible queue and is governed by normal job scheduling order. `not_before` applies to the job as a whole at creation time; it is not recalculated per attempt.

Lease Expiry and Crash Recovery

Worker health and frame ownership are separate concepts. Workers send a heartbeat every two seconds. The workers view labels a heartbeat older than 10 seconds `STALE` and older than

30 seconds `OFFLINE` . Claims by registered workers require a heartbeat within 30 seconds. These thresholds describe worker availability, but they do not directly rewrite frame state.

The frame lease is the authoritative crash detector for scheduled work. During a healthy render, two-second renewals keep its deadline in the future. If the process, host, API path, or network fails, renewals stop. The server runs `RecoverStalledWork` every five seconds and looks for `CLAIMED` or `RENDERING` rows whose `lease_expires_at` is in the past.

For each expired lease, recovery performs one transactionally consistent repair:

1. return the frame to `QUEUED` , or set it to `CANCELED` if the job is canceling/canceled;
2. clear worker ownership and lease expiry;
3. mark the current `RUNNING` attempt `FAILED` with “Worker lease expired”;
4. reset progress to `RECOVERING` at zero, or `CANCELED` for canceled work;
5. move the job to `QUEUED` , or finish it as `CANCELED` ; and
6. emit `frame.recovered` with the abandoned worker and attempt.

The recovery loop and claim path overlap intentionally. A quiet system still recovers work every five seconds. A busy system can make a simply expired row claimable immediately before selecting the next candidate. The richer recovery function is the path that closes attempt history and emits the recovery event. Operators should therefore use attempt and event records, not only the latest frame error string, when investigating a crash.

Lease duration determines the failure-detection window. It must comfortably exceed normal renewal jitter and server latency. With the API's current 60-second lease and two-second renewal interval, several missed renewals are tolerated. Recovery begins after expiry and the next recovery-loop pass, rather than exactly when worker heartbeat health changes.

Frame and Job Controls

Individual frame cancellation behaves differently for waiting and active work. A queued frame can be changed immediately to `CANCELED` , with its completion timestamp and progress updated. An active frame cannot be assumed stopped merely because a user clicked cancel. Instead, the store upserts a `frame_controls` row with `cancel_requested = 1` and changes progress to `CANCELING` . The next renewal is rejected, the worker cancels Blender, and its failure report observes the durable control row and settles the frame as `CANCELED` .

An individual retry is accepted only from `FAILED` or `CANCELED` . It clears ownership, lease, timestamps, output metadata, and error text; removes cancellation intent; resets progress; changes the job to `QUEUED` ; and resets finalization to `WAITING` . It deliberately preserves the cumulative attempt counter and old attempt rows. The next successful claim therefore has attempt $n + 1$, not attempt one.

Whole-job requeue is accepted only from `FAILED` or `CANCELED`. It resets every failed or canceled frame to `QUEUED` while leaving already complete frames untouched. This is important for large jobs: a single bad frame does not force successful frames to render again. Frame controls and progress are reset, `completed_frames` is recomputed, and finalization returns to `WAITING`. If no frame actually needs resetting and all frames are already complete, requeue instead restores `FINALIZING` and queues the finalizer.

Pause is graceful. Pausing a queued job moves directly to `PAUSED`. Pausing a running job moves to `PAUSING`; existing frame leases may renew because renewal permits both `RUNNING` and `PAUSING`, but no new claims are selected because claim eligibility includes only `QUEUED` and `RUNNING`. When an active frame completes, aggregate processing moves the job to `PAUSED`. Resume changes `PAUSED` back to `QUEUED`.

Cancel is terminal for unfinished work. It immediately marks queued and retrying frames canceled. If active frames remain, the job becomes `CANCELING`; otherwise it becomes `CANCELED`. Renewals reject canceling jobs, so workers stop and report failure, which settles those frames as canceled. Waiting or queued finalization is canceled too. Completed frames remain complete, preserving valid output and audit history.

Finalizer State

Rendering frames and packaging artifacts are different failure domains, so finalization has its own durable task and lease. A normal job starts with finalization `WAITING`. When the last frame transaction confirms a complete, failure-free, cancellation-free frame set, it changes the job to `FINALIZING`, changes finalization to `QUEUED`, and emits `finalizer.queued`.

The finalizer claim first requeues any expired `PROCESSING` finalization lease. It selects the oldest queued finalization whose job is `FINALIZING`, increments its attempt, and conditionally changes `QUEUED` to `PROCESSING`. The lease includes ordered complete frame paths, FPS, output path, and a render report. Claiming fails if the number of complete paths does not equal the requested inclusive frame count, preventing a partial sequence from becoming a final video.

On success, artifact rows and retention lifecycle rows are inserted, finalization becomes `COMPLETE`, and the job becomes `COMPLETED` in one transaction. On failure, finalization returns to `QUEUED` until its maximum attempt is reached; then finalization becomes `FAILED`, the job becomes `FAILED`, and notifications are queued. As with frame leasing, a conditional status change prevents two claimers from both acquiring the same durable task, although backend-specific contention still matters.

Sub-frame Progress and Live ETA

Workers stream bounded log batches while they own an active frame. The store persists the logs and recognizes several Blender milestones: starting, loading the blend, rendering, saving,

and validation. It also parses `Cycles Sample current/total` and `Eevee Rendering current/total samples` messages.

Sample progress is mapped into the rendering portion of a frame:

$$p_{frame} = 3 + 94 \times \min \left(1, \frac{sample}{samples} \right)$$

This reserves the first 3 percent for startup/loading and the final 3 percent for saving and validation. Ordinary progress upserts are monotonic: a lower parsed percentage cannot move the stored percentage backward. Explicit lifecycle resets, such as retry and recovery, use a separate reset operation that is allowed to return progress to zero.

`JobLiveStatus` combines completed-frame durations with a projection for the currently active frame. If a frame has started and reached at least 3 percent, its projected duration is:

$$t_{projected} = \frac{t_{elapsed}}{p_{frame}/100}$$

That projection is averaged with completed-frame durations. Remaining work subtracts completed frames and the visible fraction of the active frame:

$$t_{remaining} = \max \left(0, N - N_{complete} - \frac{p_{frame}}{100} \right) \times \bar{t}_{frame}$$

The estimated completion timestamp is the current UTC time plus that duration. Confidence is `CALCULATING` with no sample, `LOW` with fewer than three completed frames or only an active projection, `MEDIUM` after three completed frames, and `HIGH` after ten.

This is deliberately a live estimate, not a scheduling promise. The current implementation exposes one `CurrentFrame` and one progress fraction even when several slots render concurrently; whichever active row is encountered last supplies those fields. It also does not model queue delay, heterogeneous devices, retries, `not_before`, finalization time, or workspace contention. Those limitations should be considered before using the ETA for external deadlines.

Priority, Time Gates, and Fairness

Priority is an integer from -10 through 10. It orders jobs within the broader workspace rotation, with larger values first. `not_before` is a durable eligibility gate: no frame from the job is selected until the timestamp is null or in the past. Because both values live in `job_scheduling`, they survive API and worker restarts.

Workspace fairness is a lightweight least-recently-served policy. After a successful claim, the transaction upserts that workspace's `last_claim_at`. Future candidate ordering puts workspaces with the oldest claim first. Priority is evaluated after this workspace key, so a high-

priority job can outrank jobs in the same scheduling turn without allowing one recently served workspace to permanently dominate every older workspace.

This is best-effort fairness, not a formal weighted fair queue. Candidate inspection is limited to 100 rows before device filtering. A workspace with only incompatible candidates may be skipped by a particular worker. The workspace `render_limit` is not part of the claim query; worker slots cap each registered worker, but the current scheduler does not enforce a workspace-wide active-render limit. There are also no focused tests yet for priority ordering, `not_before`, rotation across workspaces, or starvation under a mixed-device backlog.

SQLite and PostgreSQL Concurrency

The same store methods support SQLite and PostgreSQL through placeholder rebinding and a dialect-adjusted schema. Their concurrency behavior is not identical.

SQLite is the development default. Foreign keys are enabled and the connection uses a 5-second busy timeout. Claiming starts with an update that recovers expired rows, so it enters the write path early. SQLite serializes database writes; concurrent claimers therefore tend to queue behind the writer lock. This makes the select-then-conditional-update transaction simple and dependable for a local server and modest worker count, at the cost of write throughput. Progress, heartbeat, log, claim, and completion traffic all compete for the same database writer. Long transactions should remain small, and “database is busy” errors remain possible after the timeout.

PostgreSQL permits more concurrent transactions. Two claimers can read the same queued candidate because the selection query does not use `FOR UPDATE SKIP LOCKED`. Their conditional updates prevent both from committing ownership: one changes `QUEUED` to `CLAIMED`, while the other eventually affects zero rows and reports no claim. Correctness is protected, but throughput can suffer from repeated collisions on the first ordered candidate, lock waits, or transaction retries under load. A production scaling step would be a PostgreSQL-specific claim query that locks and skips already contested rows while preserving the current ordering and compatibility filtering. Such a change needs dedicated concurrent tests; it should not be inferred from the current single-claim integration test.

The claim operation's related writes are transactional on both backends, so attempt creation, progress initialization, job start, fairness advancement, and the claim event cannot commit independently of ownership. Conditional renew, completion ownership checks, and finalizer conditional claims provide additional stale-writer protection. Nevertheless, database-level atomic state changes do not eliminate at-least-once external work: Blender and object storage are outside the database transaction.

What the Tests Establish

The focused SQLite tests exercise the scheduler's most important recovery contracts:

- `TestFrameLeaseLifecycle` verifies ordered frame claims, distinct workers receiving distinct frames, renewal, live logs, parsed sub-frame progress, completion metadata, and coexistence of complete and active frames.
- `TestProgressFromBlenderLogsSupportsCyclesAndEevee` verifies both supported sample formats and the rendering percentage mapping.
- `TestIndividualFrameCancelAndRetry` verifies that active cancellation blocks renewal, failure settles the cancellation, and retry preserves history by producing attempt two.
- `TestRequeueFailedJobResetsFailedFrames` verifies terminal failure followed by whole-job requeue and a second attempt.
- `TestRecoverStalledWorkRequeuesExpiredLease` creates an already expired lease, verifies `RECOVERING` progress, and confirms that the abandoned attempt becomes `FAILED`.
- `TestLiveJobStatusProjectsSingleFrameProgress` verifies a 50-percent active projection, approximately ten seconds remaining after ten seconds elapsed, and low confidence.
- `TestPauseResumeAndCancelJob` verifies graceful pause, paused claim exclusion, resumed eligibility, canceling renewal rejection, and terminal frame states.
- `TestFinalizationLifecycle` covers frame completion through finalizer claim, artifact creation, and job completion.
- Worker tests verify heartbeat-derived `STALE` and `OFFLINE` health, drain behavior, slot capacity, and device matching.

The PostgreSQL integration test is opt-in through `FRAMESHIFT_TEST_POSTGRES_URL`. It verifies migration and a complete single-worker path from job creation through frame claim, completion, finalization, artifact persistence, and publishing. It establishes dialect compatibility for that lifecycle, but it does not simulate simultaneous claims.

The current suite does not establish collision behavior under many concurrent claimers, fairness across workspaces, priority and `not_before` ordering, finalizer lease expiry, or scheduler behavior under sustained SQLite lock contention. Those are the next tests to add before increasing worker concurrency substantially. Useful tests would synchronize multiple goroutines at the claim boundary, assert that every task has at most one active attempt, run the same scenario against both databases, and verify eventual distribution rather than relying only on the first returned frame.

Operational Invariants

The scheduler is easiest to reason about through a short set of invariants:

1. The database, not a worker process, owns authoritative job and frame state.

2. Each logical frame has exactly one durable task row per job.
3. Only a conditional transition from `QUEUED` can create a lease.
4. A worker may mutate active work only while task ID, worker ID, task state, job state, and control intent still agree.
5. Every successful claim increments and records an attempt; retries never erase previous attempt history.
6. Expired leases make unfinished work recoverable, while stale workers lose the ability to renew or report over a reassigned task.
7. Job terminal and finalizing states are derived from all frame rows, not accepted from a worker.
8. Finalization is independently leased and cannot begin with an incomplete frame set.
9. Progress may advance monotonically within an attempt but can reset for an explicit retry or recovery.
10. Scheduling order combines eligibility time, workspace recency, priority, age, frame order, worker slots, and device compatibility.

These invariants allow FrameShift to recover from ordinary process and machine failures without maintaining a separate broker. They also define the boundary of the current implementation: durable and inspectable local scheduling today, with clear database and test work required before treating it as a high-contention distributed queue.

The scheduler defines ownership; the worker must now turn a temporary lease into a carefully supervised native Blender process.

Part 6 — Building the Workers: Blender, Apple Silicon, and Real-Time Observability

Rendering is where FrameShift stops being an ordinary web application and starts controlling a large native process. A frame may consume seconds or hours, allocate gigabytes, exercise a GPU, write a large file, and fail after producing convincing but incomplete output. The worker therefore has two jobs: run Blender predictably, and continuously provide enough evidence for the Go backend to decide what happened.

That distinction matters. The worker executes work, but it is not the system of record. It cannot declare a frame complete merely because Blender exited with code zero. It submits a validated result while it still owns a lease; the server decides whether that result is accepted. Claims, renewals, streamed logs, uploaded outputs, failures, and completions are all scoped to a task ID and worker ID. This keeps scheduling authoritative even when a Mac sleeps, a process crashes, or a network disappears at the least convenient moment.

Why the worker is a separate Go command

FrameShift builds the worker as `cmd/worker`, separate from `cmd/server`. The two commands share domain types and rendering packages, but they have different failure domains and deployment shapes.

The server owns durable state, API authorization, scheduling policy, and storage paths. The worker owns local process execution. Combining them would let a Blender crash, memory spike, or blocked native call compete directly with the HTTP control plane. Separation also makes remote workers natural: a Mac mini on the same desk and a machine across a private network use the same worker protocol, while only their scene and output transport differs.

Go is useful here for less glamorous reasons too. A worker is mostly concurrency, HTTP, process supervision, files, timers, and signals. Goroutines map cleanly to one heartbeat loop plus one claim loop per render slot. `context.Context` carries shutdown and lease loss into the Blender subprocess. A single binary is easy to launch with `launchd`, from a terminal, or through another service manager on macOS.

At startup, the command resolves configuration from environment variables, checks Blender with `--version`, asks Blender which Cycles devices it can see, installs signal handling, and starts its loops. The default executable path is the native application binary:

```
/Applications/Blender.app/Contents/MacOS/Blender
```

Launching that binary directly is different from opening `Blender.app` through Launch Services. It gives Go a real child process with `stdout`, `stderr`, an exit status, and a process ID. Those are prerequisites for cancellation, log streaming, and per-render resource measurements.

A lease-driven process

Each configured slot repeatedly asks the server for one frame. A claim returns either `204 No Content` or a lease containing the task, frame, resolved render settings, paths, attempt number, and expiration time. The backend filters candidates against the worker's recently advertised device list and slot capacity. A worker that advertises only CPU will not receive a frame resolved for a Metal device.

The flow for one local frame is:

Display Mermaid diagrams in this vault?

Only allow if you trust this vault's contents.

Allow

```
sequenceDiagram
    participant W as Worker slot
    participant S as Go server/store
    participant B as Blender
```

```

participant F as Filesystem

W->>S: Claim(worker ID)
S-->>W: Frame lease + resolved settings
W->>S: Renew lease
W->>B: Start native process with argument array
loop While Blender runs
    B-->>W: stdout/stderr lines
    W->>S: Batched live logs
    W->>S: Renew lease every 2 seconds
    W->>W: Sample Blender CPU and RSS
end
B->>F: Write PNG into temporary directory
W->>F: Decode header, hash, chmod, rename
W->>S: Complete(validated metadata, timing, peaks)
S-->>W: Completion accepted

```

The initial renewal immediately after a claim is intentional. It moves the task into rendering state and catches an already-invalid lease before scene preparation or Blender startup. During rendering, another goroutine renews every two seconds. The server currently grants a 60-second lease, leaving substantial room for short pauses or request jitter.

Renewal failure cancels the render context. This is more than a connectivity warning: it means the worker can no longer prove that it owns the frame. Continuing would risk two workers rendering and publishing the same task after recovery reassigned it. The correct local response is to stop Blender and report no authoritative completion.

The server independently scans for expired leases every five seconds. An expired active task is marked as a failed attempt and returned to the queue, or canceled if its job was already canceling. The next claim increments the attempt number. Crash recovery therefore does not depend on a dying worker successfully sending a final message.

Invoking Blender without a shell

The renderer uses `exec.CommandContext` with an explicit executable and argument array:

```

command := exec.CommandContext(renderContext, blenderPath,
    "--background", "--factory-startup", "--disable-autoexec", scenePath,
    "--python-expr", settingsScript,
    "--render-output", outputPattern,
    "--render-format", "PNG",
    "--render-frame", frameNumber,
)

```

No shell parses this command. Spaces in an application path, scene filename, camera name, or output directory remain data rather than syntax. There is no interpolated command string in which a quote or semicolon can acquire a second meaning. This is both a correctness property and an important boundary for untrusted uploads.

`--background` makes Blender noninteractive. `--factory-startup` reduces dependence on a user's saved preferences. `--disable-autoexec` is essential because uploaded `.blend` files are untrusted: `FrameShift` must not automatically execute scripts embedded in a scene. The small `--python-expr` is generated by `FrameShift` itself and applies resolved settings such as engine, dimensions, camera, samples, transparency, denoising, and selected Cycles device.

The generated expression uses Python string quoting for values rather than concatenating raw names. It is deliberately a single expression separated by semicolons, avoiding indentation-sensitive script construction. This does not make arbitrary Python safe; the safety comes from never accepting the expression from the upload or API client.

The renderer wraps the parent context in a per-frame timeout. The default is 30 minutes, although the `Renderer` type permits a caller to set another duration. `CommandContext` ties expiration, shutdown, or lease loss to the native process. If Blender cannot be started, startup is recorded with exit code `-1`. If it exits unsuccessfully, the renderer preserves duration, exit code, peak telemetry, parsed logs, and a bounded tail of sanitized native output in the error.

Paths are sanitized before logs leave the worker: the scene path becomes `<scene>` and the output directory becomes `<output>`. This avoids leaking machine-specific usernames and storage layouts into durable logs. It also makes messages from local and remote workers easier to compare.

Temporary output and proof of a frame

Blender never writes directly to the final frame path. The renderer creates a hidden temporary directory beside the destination and gives Blender an output pattern such as `frame_#####`. For frame 42, the expected artifact is `frame_000042.png` inside that directory.

After a successful process exit, `FrameShift` still checks for Python tracebacks and syntax or indentation failures in Blender's output. It then opens the expected PNG and calls `png.DecodeConfig`. A file is accepted only when it has a decodable PNG header and positive dimensions. The file is rewound, streamed through SHA-256, and counted to obtain its checksum and byte size.

Only after validation does the renderer set mode `0640` and rename the file to the lease's final output path. Because the temporary directory is on the same filesystem as the destination, rename provides an atomic publication boundary under normal local filesystem semantics. Consumers see either the previous state or the fully validated frame, not Blender's partially

written file. Deferred cleanup removes the temporary directory on success, failure, cancellation, or timeout.

Validation is intentionally narrower than artistic correctness. It proves that the expected output exists and is structurally a PNG of known dimensions; it does not prove that the camera was right, textures were found, the frame is nonblack, or denoising looked good. Those concerns belong to scene review, calibration renders, and future image-level checks rather than being inferred from exit code alone.

Logs become progress, not authority

Blender writes operational truth to stdout and stderr while it works. FrameShift attaches one synchronized writer to both streams. The writer stores complete output for final diagnostics and emits each complete line immediately. It also converts carriage returns to line boundaries because native progress displays often redraw one terminal line rather than appending conventional newlines.

The worker classifies lines as informational, warning, or error using conservative text matching and sends them through a buffered channel. A delivery goroutine flushes up to 20 entries at a time or every 500 milliseconds. The server accepts at most 100 entries per call and verifies that the sending worker still owns an active lease. If live delivery fails, the renderer retains its parsed Blender logs in the eventual completion or failure payload, capped server-side for durability. Thus a transient logging failure does not automatically fail a render, but it also does not allow logs from an old owner to mutate the current attempt.

Progress interpretation lives in the backend. Cycles commonly emits lines resembling:

```
Fra: 1 | Remaining: 00:10 | Sample 25/100
```

Eevee can emit:

```
Fra: 1 | Rendering 32 / 64 samples
```

The store recognizes both forms with regular expressions. Samples occupy the range from 3 to 97 percent:

$$p = 3 + 94 \min \left(1, \frac{s}{S} \right)$$

where s is the current sample and S is the total. The reserved edges describe startup, loading, saving, and validation. Recognized messages such as “Starting Blender,” “Read blend,” “Saved,” and “rendered and validated” move the phase through `STARTING`, `LOADING`,

RENDERING , SAVING , and VALIDATING . Progress updates are monotonic in storage, so a late or repeated line cannot move a frame backward.

This parser is necessarily heuristic. Blender's text format is not a stable structured telemetry protocol, and engines or Blender versions may change wording. Unknown lines remain useful logs but do not invent progress. The tradeoff is honest partial observability instead of false precision.

ETA from completed and active evidence

The live job status combines completed frame durations with a projection for the active frame. Once active progress reaches at least 3 percent, projected duration is:

$$t_{projected} = \frac{t_{elapsed}}{p/100}$$

The backend averages that projection with completed-frame durations. Remaining work subtracts both completed frames and the active frame's fractional progress, then multiplies by the current average. It reports the resulting duration and estimated completion timestamp.

Confidence is CALCULATING before any useful sample, LOW with an active projection or fewer than three completed frames, MEDIUM after three completed frames, and HIGH after ten. This is a practical estimate, not a promise. Frames can differ dramatically due to geometry, simulation state, motion blur, shader compilation, or asset caching. Multiple slots also make a single-active-frame model conservative or approximate; a future ETA model could account explicitly for every active lease and worker throughput.

CPU, RSS, and the Apple GPU gap

There are two related telemetry paths. The heartbeat samples the Go worker process itself: CPU percentage and resident set size (RSS). This shows whether the coordinator is alive and behaving reasonably, but it does not include all Blender memory in the worker's RSS because Blender is a separate process.

During each render, the renderer obtains Blender's PID and samples that process every 200 milliseconds with `gopsutil`. It records peak CPU percentage and peak RSS in the attempt metrics. These are sampled peaks, so a spike shorter than the interval may be missed, and CPU percentage can exceed 100 on a multicore interpretation. Still, they are useful for comparing frames, finding memory-heavy scenes, and diagnosing workers that fail under concurrency.

Host telemetry separately reports CPU model, logical and physical cores, memory, and filesystem usage. On NVIDIA systems, FrameShift uses `nvidia-smi` to read device name, GPU utilization, and memory used and total. There is no equivalent unprivileged, stable

command used here for Apple GPUs. On macOS, `powermetrics` generally requires privileges and its output and availability are not a suitable dependency for the normal worker. FrameShift therefore reports Apple GPU utilization as unavailable with the explicit reason that privileged `powermetrics` is not sampled.

That limitation must remain visible. Metal support does not imply that FrameShift can report “73% GPU” or unified-memory pressure with NVIDIA-like fidelity. CPU/RSS, frame duration, and Blender logs are observable; Apple GPU utilization is not currently measured. Fabricating a number from total CPU or render speed would be worse than an unavailable field.

Apple M4 and Metal discovery

FrameShift does not guess Metal capability from the Mac model name. It starts Blender in a short, 20-second discovery command, loads the Cycles preferences, calls `get_devices()`, and prints a JSON payload behind a unique `FRAMESHIFT_DEVICES=` marker. The Go parser ignores Blender's surrounding startup output and decodes entries with Blender's device ID, display name, and type.

On an Apple M4 machine, Blender may return a CPU device named for the M4 and a GPU device whose type is `METAL`. Exact IDs and names belong to Blender and can vary, so the scheduler matches the resolved `DeviceType` and, when specified, `DeviceID` against the advertised list rather than hard-coding `METAL_M4` as universal truth.

For a Cycles frame resolved to Metal, the generated settings expression selects `compute_device_type = "METAL"`, refreshes devices, enables only the matching device ID, and sets `scene.cycles.device = "GPU"`. CPU work remains the default when no accelerator is requested. Eevee and Workbench do not use the same Cycles preference path.

Discovery failure is nonfatal at worker startup: the worker logs the error and advertises a CPU fallback. Blender version discovery, by contrast, is fatal because a worker with no runnable Blender cannot fulfill its purpose. This asymmetry lets an otherwise healthy Mac keep rendering CPU-compatible jobs when a Metal preference or driver probe fails.

Slots, draining, and health

`FRAMESHIFT_WORKER_SLOTS` controls the number of independent claim loops. Each loop processes at most one lease, so a worker with four slots may run four Blender processes. An atomic active counter turns the heartbeat status from `AVAILABLE` to `BUSY` when any slot is occupied.

Slots are a concurrency ceiling, not a promise that the machine can sustain that many renders. On Apple Silicon, CPU and GPU share memory. Two Metal renders can contend for the same GPU and unified-memory bandwidth; several CPU renders can saturate performance cores and trigger memory pressure. Operators should begin with one slot, measure representative frames,

and increase only when aggregate throughput improves without destabilizing the host. The server also checks advertised slot count and active leases before granting more work.

Drain is server-owned control. Setting a worker to `DRAINING` makes future claims return no work while existing leases continue to renew and finish. The worker does not need a special command channel; its ordinary polling observes the scheduling decision. Resume restores claims. Removal is allowed only when no claimed or rendering tasks remain. This makes maintenance safe: drain, wait for active leases to reach zero, then stop or remove the process.

Heartbeats arrive every two seconds. The registry reports a worker `ONLINE` when the last heartbeat is no more than ten seconds old, `STALE` after ten seconds, and `OFFLINE` after thirty. `DRAINING` takes precedence as an operator-visible state. Scheduling requires a heartbeat newer than thirty seconds, so an old process record cannot claim work merely because it still exists in the database.

Signals, crashes, and the macOS experience

The worker listens for `SIGINT` and `SIGTERM` through `signal.NotifyContext`. Pressing Control-C in Terminal, stopping a service, or asking `launchd` to terminate the process cancels the shared context. Claim loops stop waiting, heartbeat work ends, active render contexts are canceled, and the worker waits for its goroutines before returning.

On macOS this direct process model is pleasantly unsurprising: Blender's executable is inside the application bundle, POSIX signals reach the Go process, and normal absolute paths work with `exec.CommandContext` without shell escaping. Files are created with restrictive modes, render directories use `0750`, intermediate files use private temporary locations, and published frames use `0640`.

There are still operational edges. App translocation, quarantine prompts, privacy controls, network-volume behavior, sleep, and abrupt power loss sit below the application. Atomic rename is strongest when staging and destination share a local filesystem, which is why local staging is created beside the destination. A force kill or machine crash gives the worker no opportunity to send failure; lease expiration and server recovery handle that case. If macOS sleeps longer than the lease, the task may be reassigned even if the original Blender process later wakes. Its stale completion will be rejected because ownership changed.

Network workers and scene bundles

Local workers receive server-resolved paths under the shared storage root. A remote worker cannot dereference those paths, so `FRAMESHIFT_WORKER_REMOTE=true` changes preparation and publication.

After claiming and renewing, the worker downloads a ZIP for the leased scene version. The server bundles regular files under that version directory, skips symlinks, and returns the relative

`.blend` path in a header. The worker limits the response stream, rejects absolute or nonlocal paths, backslashes, and symlink entries, and creates each extracted file with `0_EXCL` under a fresh temporary directory. Blender renders there, and cleanup removes the bundle afterward.

The output travels back with an authenticated `PUT` as `image/png`. The server verifies lease ownership, streams into a bounded temporary file, validates the PNG again, applies permissions, and atomically renames it into managed storage. The server's validation result becomes the checksum, dimensions, and size submitted for completion. This second validation is valuable because transport is another trust boundary.

Bundling a scene version does not magically make every Blender dependency portable. Packed assets and files ingested into the managed version directory travel well. Absolute texture paths, linked libraries outside the version root, fonts, caches, simulation data, add-ons, and environment-dependent color-management configuration may not. Remote assets should be normalized or packed during ingestion, and unresolved dependencies should block production work or be surfaced clearly in review. Secretly reaching back to arbitrary workstation paths would make renders irreproducible and create a serious network and security concern.

The current remote transport also favors simplicity over deduplication: a worker can download a version bundle per lease. For large scenes, future work could use content-addressed blobs, cache manifests, resumable downloads, and checksums before extraction. Those optimizations must preserve the same rule: the server identifies the approved source, and the worker treats every received archive as untrusted.

Alternatives and tradeoffs

One alternative is embedding Blender's Python runtime or running a persistent Blender daemon. That could reduce startup time and retain caches, but it couples attempts inside one long-lived address space. Scene state, add-ons, leaks, and crashes can contaminate later frames. One native process per frame costs startup time but gives FrameShift a strong cleanup and isolation boundary.

Another option is shelling out through a command string. It is shorter in a prototype, but quoting becomes platform-dependent and unsafe as soon as paths or user-derived values enter the command. Argument arrays are the more durable interface.

A shared network filesystem could replace bundle download and output upload. It avoids transfer code and may be appropriate on a trusted render LAN, but it introduces mount identity, permissions, latency, partial writes, and weaker isolation between workers. FrameShift supports the straightforward local path and an explicit HTTP remote path rather than pretending they have identical failure modes.

Finally, progress could be omitted until each frame completes. That would be simpler and more stable than parsing native text, but long frames would look hung and ETA would have no active evidence. FrameShift accepts a modest parser maintenance cost while keeping parsed progress advisory. The lease and validated artifact, not a progress line, decide completion.

The worker architecture is intentionally conservative: native processes are disposable, outputs are untrusted until decoded and hashed, observability is streamed but not authoritative, and the backend can recover without cooperation from a failed Mac. That foundation matters more than squeezing every possible frame per second from the first implementation. Once execution and ownership are dependable, performance tuning becomes measurable rather than hopeful.

Validated frames are durable evidence, but users still need a complete, inspectable, and deliverable artifact set.

Part 7 — From Frames to Deliverables: FFmpeg, Artifacts, and Publishing

A directory full of rendered PNG files is valuable, but it is not yet a finished delivery. Artists usually need a video they can review immediately, a smaller preview that behaves well in a browser, a representative thumbnail, the original frames, and enough machine-readable evidence to explain where every output came from. Operations needs a different set of guarantees: interrupted packaging must recover, incomplete files must not appear as completed outputs, retention must be enforceable, and publishing to another system must not weaken authorization or network security.

FrameShift treats those requirements as a distinct finalization and delivery pipeline. Blender produces individual frames, while FFmpeg and FFprobe turn an accepted frame sequence into media. The Go backend remains the system of record throughout. A finalizer may create bytes, and a publisher may copy or announce them, but only store transactions decide that finalization completed, which artifacts exist, which storage tier owns them, and whether an external delivery is approved.

This separation matters because rendering and delivery fail differently. A Blender worker can finish every frame successfully while FFmpeg is unavailable. A video can be encoded while its preview fails. All local outputs can be valid while a webhook is down. Combining these facts into one optimistic “done” flag would make recovery ambiguous. FrameShift instead exposes `FINALIZING`, artifact lifecycle states, and publish request states as durable facts.

The complete data flow

The path from a worker's final frame report to a recipient is deliberately longer than a direct file copy:

Display Mermaid diagrams in this vault?

Only allow if you trust this vault's contents.

Allow

flowchart LR

```
W[Render worker] -->|complete frame with checksum and dimensions| S[(Authoritative store)]
S -->|aggregate all frame rows| A{Frame set outcome}
A -->|pending remains| R[Job remains running or paused]
A -->|failed or canceled| T[Terminal job state]
A -->|all complete| Q[Job FINALIZING and task QUEUED]
Q -->|durable lease| F[Finalizer]
F --> M[FFmpeg final video and preview]
F --> P[FFprobe and PNG validation]
F --> Z[ZIP, report, manifest, SHA-256]
M --> X[Temporary artifact directory]
P --> X
Z --> X
X -->|promote six files| L[(Managed storage)]
L -->|transaction: artifact rows and lifecycle rows| S
S -->|job COMPLETED| UI[Authenticated output UI]
UI -->|inline or attachment| D[Authorized user]
S -->|approved durable publish request| O[Publish outbox loop]
O -->|atomic local copy| PF[(Published folder)]
O -->|signed event| H[Webhook recipient]
L <-->|HOT / COLD transition| C[(Local archive or MinIO)]
```

The arrows into the database are as important as the media operations. Filesystem work and remote HTTP requests cannot participate in the SQL transaction, so FrameShift arranges the order to avoid claiming success too early. Large bytes live in managed storage; rows give those bytes identity, authorization, retention, and meaning.

Completion is an aggregate, not a worker claim

A worker completes one leased frame task by reporting evidence such as its SHA-256 checksum, byte size, dimensions, duration, resource measurements, and bounded logs. It cannot mark the parent render complete. In the same transaction that accepts the current lease owner's frame result, the store reloads an aggregate over every durable frame task for the job.

The aggregate asks how many tasks are still pending and how many finished as complete, failed, or canceled. State precedence is intentional:

1. If pending work remains, the job continues running, or settles into `PAUSED` when a pause was already requested.
2. If nothing is pending and any frame failed, the job becomes `FAILED`.

3. If nothing is pending and any frame was canceled, the job becomes `CANCELED` .
4. If every task is complete, a calibration job completes directly because it does not produce the normal deliverable set.
5. If every task is complete for a production render, the job becomes `FINALIZING` and its finalization task changes from `WAITING` to `QUEUED` .

The last transition and the `finalizer.queued` event commit with the accepted frame completion. There is no interval in which the frame is durably complete but the application merely hopes an in-memory queue received a signal. A polling finalizer can always rediscover queued work after a restart.

This aggregate rule also closes a common distributed-rendering race. Several workers can finish near the same time, but no worker knows whether another frame is still leased, retrying, failed, or being canceled. The database transaction sees the complete set and derives the parent state. A file appearing in the frames directory is not sufficient evidence: only a current lease owner's committed result counts.

Before leasing finalization, the store loads complete frame paths ordered by frame number and requires their count to equal the inclusive range:

```
expected := lease.FrameEnd - lease.FrameStart + 1
if len(lease.FramePaths) != expected {
    return domain.FinalizationLease{}, false,
        fmt.Errorf("finalization frame set is incomplete")
}
```

The finalizer independently verifies that every resolved path is local to managed storage, exists, and matches the contiguous naming convention `frame_%06d.png` . The store establishes completeness; the service establishes that the corresponding files are still present and structurally addressable.

Durable finalization leases and retries

Finalization is expensive, can outlive an HTTP request, and invokes native child processes. It therefore has its own durable task rather than running inside the transaction that accepts the last frame. The task moves through `WAITING` , `QUEUED` , `PROCESSING` , `COMPLETE` , and `FAILED` .

The server loop claims one queued task with a five-minute lease. Claiming increments the attempt number and conditionally updates `QUEUED` to `PROCESSING` ; if another claimant already won, the affected-row check rejects the lease. At the beginning of each claim transaction, an expired `PROCESSING` task is returned to `QUEUED` with “Finalizer lease expired” recorded as its

error. This is at-least-once execution: a server crash can cause the same logical finalization to run again, but it cannot make the work disappear.

The finalizer service itself gives its FFmpeg/FFprobe work a 15-minute default context timeout. That timeout bounds native process execution, while the database lease bounds authority to commit the result. The current loop does not renew a finalization lease during a long encode. Installations with renders that can exceed five minutes should align those limits or add lease renewal; otherwise a second server could reclaim the task while the first process is still working. Deterministic output names, temporary directories, conditional store transitions, and cleanup reduce the consequences, but renewal would make the ownership model stronger.

On a service error, `FailFinalization` records the message and emits `finalizer.failed`. Attempts below three return the task to `QUEUED` while the job remains `FINALIZING`. The third failure makes both the task and job failed and queues failure notifications. On success, all artifact rows, their retention rows, the finalization `COMPLETE` state, the job `COMPLETED` state, completion notifications, and events commit in one transaction.

Retries are valuable precisely because frames do not need to be rendered again. A missing FFmpeg binary, temporary disk pressure, or a transient process failure belongs to packaging, not pixel production. Preserving that boundary saves expensive work and makes the reported failure comprehensible.

Why FFmpeg and FFprobe

Media encoding is not a useful place for FrameShift to invent a codec stack. FFmpeg has mature sequence input, color and pixel-format conversion, H.264 encoding, MP4 muxing, filtering, and broad platform availability. FFprobe supplies a separate inspection interface with structured JSON output. Go orchestrates these tools, constrains their inputs, captures bounded diagnostics, and validates their products.

Using separate executables also keeps native codec complexity out of the server's address space. A codec failure exits a child process instead of corrupting the coordinator.

`exec.CommandContext` connects cancellation and timeout to process lifetime. Most importantly, every token is passed as an argument array; FrameShift does not interpolate a shell command:

```
command := exec.CommandContext(ctx, ffmpegPath,
    "-hide_banner", "-loglevel", "error", "-y",
    "-framerate", fps,
    "-start_number", firstFrame,
    "-i", framePattern,
    "-frames:v", frameCount,
    "-vf", "pad=ceil(iw/2)*2:ceil(ih/2)*2",
    "-c:v", "libx264",
```

```
"-pix_fmt", "yuv420p",  
"-movflags", "+faststart",  
temporaryVideo,  
)
```

There is no shell to reinterpret spaces, semicolons, substitutions, or redirections. Paths are first resolved beneath the configured storage root, and frame names are generated by FrameShift rather than accepted as arbitrary user command fragments. Argument arrays are not a complete sandbox, but they remove an avoidable command-injection boundary.

The primary output uses `libx264` and `yuv420p`. H.264 in an MP4 container is a pragmatic review and delivery baseline with excellent browser and player compatibility. `yuv420p` avoids producing a chroma format many consumer decoders reject. The pad filter rounds odd frame dimensions up to even values because common 4:2:0 encoders require even chroma planes. The test suite deliberately uses 65 by 37 source frames to exercise this behavior.

`-movflags +faststart` relocates MP4 metadata toward the beginning of the file. A browser can read stream metadata and begin playback without downloading the entire object first, which matters for both local HTTP serving and object-backed artifacts. The preview is encoded from the validated primary video with no audio, the `veryfast` preset, CRF 32, `yuv420p`, and fast-start. It trades fidelity for size and encode speed while leaving `final.mp4` as the primary deliverable.

Process exit status is necessary but insufficient. After encoding, FrameShift invokes FFprobe with `-select_streams v:0`, requests codec, width, height, frame count, and format duration, and parses JSON. The final video must expose one selected H.264 video stream with positive dimensions. The preview must also expose an H.264 video stream. This catches empty, malformed, or unexpectedly encoded files before they become artifacts. The implementation records duration in the manifest, although it does not yet enforce an expected duration or frame count from the probe; stricter validation could compare those values with `(frameEnd - frameStart + 1) / fps` within a muxing tolerance.

The atomic six-artifact set

A successful production finalization returns exactly six artifact records:

Type	Filename	Purpose
VIDEO	<code>final.mp4</code>	Primary H.264 delivery
PREVIEW	<code>preview.mp4</code>	Smaller browser-oriented review encode
THUMBNAIL	<code>thumbnail.png</code>	320-pixel-wide poster image from the first frame

Type	Filename	Purpose
FRAME_ARCHIVE	frames.zip	Deflated original PNG sequence with stable basenames
REPORT	render-report.json	Operational and provenance summary
MANIFEST	artifact-manifest.json	Integrity and media metadata for the set

All six are first constructed in a `.finalize-*` temporary directory inside the job's artifact directory. The finalizer validates the main video and preview with FFprobe, decodes the thumbnail header as PNG, closes the ZIP cleanly, marshals both JSON documents, computes every SHA-256 digest and byte size, and applies mode `0640`. Only then does it rename each candidate to its deterministic final name.

If any rename fails, deferred compensation removes every final path that may already have moved. This is stronger than exposing files one by one, but “atomic” needs a precise definition. Six independent renames are not a single filesystem operation, and the filesystem cannot join the subsequent SQL transaction. A process or machine crash at the wrong instant can leave orphaned files. Normal application readers still cannot discover them through the artifact API because no artifact rows exist until `CompleteFinalization` commits. The set is atomic at the authoritative application boundary: readers observe either no committed artifact set or all rows plus a completed job. A reconciliation task can later remove unreferenced artifact files after a grace period.

The SHA-256 checksum on each artifact serves several purposes. It gives clients an integrity value through the `X-Checksum-SHA256` response header, lets publication identify the exact video announced, and makes backup or migration verification possible without trusting filenames or modification times. A checksum is evidence about bytes, not authenticity by itself; authenticity comes from obtaining the expected digest through an authorized FrameShift API or signed message.

The manifest binds media facts to the set. It includes job ID, frame range and count, FPS, codec, padded dimensions, probed duration, creation time, and checksum/size entries for the video, preview, thumbnail, frame ZIP, and report. The manifest's own checksum lives in its artifact row because a document cannot contain its final digest without a recursive definition. Versioning the manifest schema would be a useful next step before external consumers depend on it.

The render report answers operational questions rather than merely listing files. It carries the job and frame range, completion and failure counts, retry count, total and average frame duration, output bytes, peak memory and CPU observations, Blender version, render engine, and fastest/slowest frames where available. The finalizer stamps it `COMPLETED` and adds a

generation time. This keeps performance and provenance information beside the media even if the live database is later unavailable.

The frame ZIP preserves the accepted PNG sequence for compositing, re-encoding, or forensic comparison. Entries use only generated frame basenames and mode `0640` ; they do not reproduce arbitrary directory trees. ZIP is widely interoperable and streamable to standard tooling, though creating it currently duplicates all frame bytes and can temporarily require substantial disk space. For very large sequences, a streamed archive, ZIP64-aware limits, or an optional “do not archive frames” policy may be warranted.

Authenticated downloads and the output UI

Artifacts are not exposed as static files. Listing a job's artifacts joins the job through project membership. Fetching an artifact by ID repeats that authorization check, so knowing an identifier does not grant access. The API provides separate attachment and inline routes. Both set the recorded MIME type, a sanitized `Content-Disposition` , and `X-Checksum-SHA256` ; local files are served with Go's `http.ServeContent` , which also supports normal media range behavior.

Object-backed artifacts pass through the same authenticated endpoint. An `object://` storage path is parsed into an object key, opened through the configured client, and streamed by the server. FrameShift does not hand the browser a permanent MinIO URL or leak host filesystem paths. Individual completed frames have their own authenticated download route and use the frame result's checksum.

The web output gallery selects the latest completed job. It uses `preview.mp4` , falling back to the main video, in a credentialed HTML video player and uses the thumbnail content endpoint as its poster. The delivery list shows all six artifact names, types, sizes, and hot/cold state. Each row offers pin/unpin, archive/restore, and download actions. The first 24 individual frame numbers are exposed as direct download controls, while `frames.zip` provides the complete sequence. The queue view also offers quick downloads for the primary video and manifest.

This UI is a projection of backend facts. It waits for `COMPLETED` before loading artifacts and does not infer completion from 100 percent frame progress. Lifecycle actions send intent to the API; they do not rewrite local state optimistically unless the server returns the updated artifact. Publish requests and their durable statuses are polled while active, making `REQUESTED` , `APPROVED` , `DELIVERING` , `RETRY` , `DELIVERED` , and `FAILED` visible rather than hiding external delivery behind a button spinner.

HOT, COLD, pinning, and retention

Every artifact receives a lifecycle row in the same transaction that publishes it. New artifacts start `HOT` , unpinned, with `retain_until` calculated from the configured artifact retention

period. The default is 90 days; administrators may configure between 1 and 3650 days. A cleanup operation selects only unpinned artifacts whose retention deadline is in the past, removes the underlying local file or object, and then deletes the row.

`HOT` means the artifact is in its original managed job path and ready for ordinary low-latency access. `COLD` is an explicit placement decision, not shorthand for “deleted.” Without object storage, cold transition renames the file beneath `storage/archive/<artifact-id>/`; restoring it moves the file back to its recorded original path. With object storage configured, cold transition uploads to `artifacts/<artifact-id>/<filename>`, updates the path to `object://...`, then removes the hot copy. Restore uses a temporary download and rename before deleting the object.

The API performs byte movement before changing the database lifecycle row and compensates when that row update fails. This again is not a distributed transaction: crashes can create duplicates or orphans, so periodic reconciliation is appropriate for installations that rely heavily on tiering. The authoritative row always tells normal readers which copy to use.

Pinning protects valuable outputs from retention cleanup regardless of their tier. It is independent of `HOT` and `COLD`: a pinned cold archive remains retained, and an unpinned hot file can eventually expire. Approving publication automatically pins the video, preview, thumbnail, and manifest so the review surface and integrity metadata associated with an external delivery are not immediately cleaned up. The report and frame ZIP are not automatically pinned by that operation, a policy tradeoff worth reviewing for archival or regulated workflows.

MinIO is optional because local-first use should not require an object service. When configured, `minio-go` handles S3-compatible authentication, bucket creation, upload, download, streaming, and removal. MinIO gives development and private deployments the same API shape as many hosted object stores without changing FrameShift's lifecycle authority. The tradeoff is operational: credentials, TLS, bucket durability, object retention, and backups become part of the installation. S3 compatibility also does not guarantee identical behavior across every vendor, so integration tests should cover the selected target.

Local publishing and approval-gated destinations

Downloading is a user pull operation. Publishing is a controlled push toward a configured destination, so it has a separate permission and approval model. A workspace administrator creates either a `LOCAL_FOLDER` destination or an HTTP(S) `WEBHOOK`. A user with job-control capability may request publication only for a completed job and only to a destination in the job's workspace. The request begins as `REQUESTED`; an owner or administrator must explicitly approve it before delivery becomes claimable.

Approval is durable, audited, and separate from request creation. This two-step design prevents a compromised artist session or accidental click from immediately sending an output outside the workspace. Destination creation, publication request, and approval all produce durable audit or event records. Destinations with active requests cannot be deleted out from under delivery.

For a local-folder destination, the target is a single safe path segment, not an arbitrary absolute path. Delivery copies the primary `final.mp4` to a temporary file under:

```
storage/published/<destination>/<job-id>/final.mp4
```

After copy and close succeed, FrameShift applies mode `0640` and renames the temporary file into place. The deterministic destination makes retries idempotent at the file level: a retry replaces the same logical output rather than creating numbered duplicates. The current publisher sends only the `VIDEO` artifact, not the entire six-file artifact set. The webhook similarly announces the video's filename, checksum, and size rather than transferring media bytes.

Durable outbox delivery and idempotency

The `publish_requests` table acts as a durable outbox. The server loop polls for `APPROVED` or `RETRY` requests, requeues expired `DELIVERING` leases, conditionally claims one request for a minute, and increments its attempt. A successful external action changes the row to `DELIVERED` ; an error returns it to `RETRY` until the fifth attempt, after which it becomes `FAILED` .

This is intentionally at-least-once delivery. Consider a webhook that accepts the POST just before FrameShift crashes. The request remains `DELIVERING` ; after lease expiry it is sent again. Exactly-once remote effects are impossible unless the receiver cooperates. FrameShift therefore sends the stable publish request ID in the `Idempotency-Key` header and as payload `id` . A receiver should persist that key with its result and return the previous success for duplicates.

Local publication has analogous behavior. Rename gives readers a complete file, and retrying writes through a new temporary file to the same final name. There can still be a crash after rename but before the database marks `DELIVERED` ; repeating the copy is harmless because the content is identified by the same artifact checksum. Downstream automation should compare that checksum if replacement semantics matter.

The current retry loop has no increasing backoff or `next_attempt_at` ; it can retry a persistently failing destination as quickly as the claim loop allows. Exponential backoff with jitter and an operator-triggered replay action would reduce pressure and improve recovery. The durable state model already provides the right place to add those policies.

HMAC webhooks, SSRF, and redirect boundaries

Webhook bodies are compact JSON events containing the request ID, event name `render.published`, job identity, and primary artifact metadata. FrameShift computes HMAC-SHA-256 over the exact body using the configured webhook secret and sends:

```
X-FrameShift-Signature: sha256=<hex digest>
Idempotency-Key: <publish request id>
```

The receiver should compute HMAC over the raw bytes, compare signatures with a constant-time function, reject stale or unexpected event types, and deduplicate by idempotency key. HMAC authenticates the sender and body; HTTPS is still required to protect metadata confidentiality and reduce replay exposure in transit. A timestamp header and bounded replay window would strengthen the protocol further.

User-configured webhook URLs create a server-side request forgery risk because the backend makes the HTTP request from a more privileged network position. FrameShift accepts only `http` or `https` URLs with a hostname, resolves the hostname before delivery, and by default rejects private, loopback, and link-local addresses. Private targets require the explicit `FRAMESHIFT_ALLOW_PRIVATE_WEBHOOKS=true` opt-in, useful for controlled local integrations but inappropriate as a casual production default. The client also has a ten-second timeout.

Those checks are a baseline, not a complete SSRF firewall. The standard HTTP client may follow redirects to an address that was never validated. DNS can also change between the explicit lookup and the client's connection, creating a time-of-check/time-of-use or rebinding window. Production hardening should disable redirects or validate every redirect target, resolve and pin an allowed IP in a custom transport, reject unusual numeric and IPv6 forms consistently, constrain ports, and apply an outbound network policy. An allowlist of approved domains is stronger than trying to enumerate every forbidden network. Destination validation should also be repeated at delivery, as FrameShift currently does, because DNS and policy can change after configuration.

Package alternatives and tradeoffs

The implementation favors mature native tools and a small amount of explicit Go orchestration, but other choices are possible:

Concern	Current choice	Alternative	Tradeoff
Video encoding	Native FFmpeg process	GStreamer, libav bindings, platform encoders	GStreamer offers pipelines; bindings avoid processes but add C ABI, memory-safety, and distribution complexity; platform encoders reduce portability

Concern	Current choice	Alternative	Tradeoff
Media inspection	Native FFprobe JSON	go-astlav , MediaInfo, parse FFmpeg logs	Bindings expose richer structures but couple builds to native libraries; logs are not a stable data contract
Frame archive	Go archive/zip	tar.zst , external zip , streamed object multipart upload	ZIP is universally convenient; tar plus Zstandard is often faster and smaller but less friendly to desktop users
Checksums	Go crypto/sha256	BLAKE3, object-store ETags	SHA-256 is ubiquitous and stable; BLAKE3 is faster but less universal; multipart ETags are not reliable content hashes
Object storage	minio-go/v7	AWS SDK for Go v2, filesystem only, cloud-specific SDK	MinIO keeps S3-compatible setup concise; AWS SDK exposes broader AWS features; filesystem-only is simpler but lacks remote cold storage
Durable work	SQL status rows and leases	Redis queues, NATS JetStream, cloud queues	A broker can reduce polling and scale consumers, but introduces another authority and still needs idempotent database coordination
Webhook signing	HMAC-SHA-256	Ed25519 signatures, mTLS	HMAC is simple but shares one secret; asymmetric signatures ease receiver verification; mTLS strengthens transport identity but raises certificate operations

The key selection criterion is not dependency count. It is whether a package owns difficult domain behavior without becoming a second source of truth. FFmpeg owns codecs. MinIO's client owns S3 protocol details. Go's standard library is sufficient for ZIP, hashing, process argument arrays, and HTTP composition. SQL rows remain the right place for FrameShift-specific authority: leases, approval, artifact identity, retention, and delivery outcomes.

Guarantees and limits

The resulting pipeline has a clear definition of “ready.” Every durable frame is complete; a finalizer held a claim; the sequence was contiguous and present; the primary and preview videos encoded; FFprobe accepted H.264 streams; the thumbnail decoded; the archive and JSON documents closed successfully; every file was hashed; all six paths were promoted; and one transaction published all artifact rows while completing the job.

That definition is stronger than “FFmpeg exited zero” and more honest than “files probably exist.” It also leaves explicit operational work: reconcile orphaned files and objects, renew long finalization leases, add stricter duration/frame validation, version manifests, back off publication retries, and harden webhook connection routing against redirects and DNS rebinding. These are extensions of the existing authority model rather than reasons to bypass it.

Most importantly, delivery does not erase provenance. The browser streams through authenticated routes, downloads carry expected checksums, lifecycle policy stays attached to each artifact, cold storage does not change identity, and external publication begins only after durable approval. FrameShift turns frames into deliverables without turning the filesystem, browser, FFmpeg process, or webhook recipient into the system of record.

Valuable outputs introduce people and policy: identity, team roles, durable communication, and explicit production approval.

Part 8 — Identity and Collaboration: Authentication, Teams, Email, and Approval

FrameShift begins as a local rendering tool, but the moment a second person can sign in, the product acquires a different kind of state. A render job is no longer merely a command sent to Blender. It belongs to someone, consumes a shared quota, may be operated by one teammate and reviewed by another, and can produce an email that must survive a server restart. Identity and collaboration therefore belong in the same architectural story as scheduling: the Go backend remains the authority, and every client sends intent rather than declaring outcomes.

This chapter follows that authority through five connected areas: passwords and sessions, workspace membership, invitation and notification email, scene approval, and browser desktop notifications. The implementation is deliberately local-first. SQLite can hold all authoritative records, Mailpit can capture email without an external provider, and the browser can provide a useful notification surface. None of those choices weaken the central rule: a hidden button is not authorization, a displayed badge is not a state transition, and a sent SMTP message is not the source of truth.

Passwords: expensive by design

FrameShift hashes passwords with Argon2id. Password hashing has a different goal from ordinary hashing. SHA-256 is intentionally fast, which makes it excellent for checking a high-entropy random token and dangerous for protecting a human password. If a password database is copied, an attacker can test guesses offline. A password hash should make every guess consume meaningful time and memory.

Argon2id combines the side-channel resistance associated with Argon2i's data-independent access pattern with Argon2d's resistance to GPU cracking. That makes it a practical default for

an interactive login service. FrameShift currently uses a 16-byte random salt, a 32-byte result, three iterations, two lanes of parallelism, and 64 MiB of memory:

```
hash := argon2.IDKey(  
    []byte(password), salt,  
    3, 64*1024, 2, 32,  
)
```

The encoded value includes the algorithm, version, parameters, salt, and result in the familiar `$argon2id$...` format. Storing parameters with each hash matters. The application can raise the cost for new passwords later while still verifying old records, then rehash a successfully verified password when policy changes. Salts are generated with `crypto/rand`, so equal passwords do not produce equal database values. Verification reconstructs the configured calculation and uses constant-time comparison for the derived bytes.

These parameters are a deployment baseline, not a timeless constant. A production operator should benchmark them on the actual API hardware and target an acceptable interactive latency under expected concurrency. Raising memory too far can turn login bursts into self-inflicted denial of service; setting it too low makes an offline breach cheaper. Alternatives include `bcrypt`, which is mature but primarily CPU-hard and has a 72-byte password limitation, and `scrypt`, which is memory-hard but less convenient to tune. Argon2id offers the clearest modern balance for this service.

Login also avoids a common account-enumeration timing signal. When the email lookup fails, the server still verifies the submitted password against a precomputed dummy Argon2id hash. The response remains the same generic `INVALID_CREDENTIALS` result for an unknown email and a wrong password. This does not make every network timing distribution identical, but it removes the obvious fast path.

Opaque sessions and defensive cookies

FrameShift uses server-side sessions rather than putting identity or authorization claims into a self-contained browser token. A new session starts with 32 random bytes encoded for URLs. The raw token goes to the browser; only its SHA-256 digest goes into `user_sessions`:

```
cookie:      base64url(random 32 bytes)  
database:    base64url(SHA-256(cookie value))  
lookup:      hash presented cookie, then match an unexpired row
```

SHA-256 is appropriate here because the input is not a guessable password. A uniformly random 256-bit token already has overwhelming entropy, so an expensive password hash would add server cost without meaningfully improving resistance to offline guessing. Hashing

still provides an important breach boundary: a read-only database leak does not immediately reveal bearer credentials that can be replayed against a live server.

The `frameshift_session` cookie is `HttpOnly`, preventing ordinary JavaScript from reading it; `SameSite=Lax`, reducing cross-site request attachment; scoped to `/`; and limited to 24 hours. `Secure` is configuration-driven so local HTTP development works while HTTPS deployments can require transport protection. Authenticated fetches opt into credentials, while the backend resolves the user from the stored session on every protected request.

Session creation also rotates the credential. If a session cookie is already present during registration or login, its database row is deleted before a new random token is issued. That prevents a known or fixed pre-authentication token from surviving authentication. Logout hashes the cookie, deletes the matching row, and expires the browser cookie. The session-management API lists a user's unexpired sessions, marks the current one by comparing hashes, permits revocation by session ID only within that user, and supports revoking every session. Revoking the current session also clears the cookie. A successful password reset deletes every session for the account in the same transaction as the password change.

Display Mermaid diagrams in this vault?

Only allow if you trust this vault's contents.

Allow

sequenceDiagram

```
participant B as Browser
participant A as Go API
participant D as Database
B->>A: POST login with email and password
A->>D: Load password hash
A->>A: Verify Argon2id
A->>D: Delete session named by existing cookie
A->>A: Generate random token and SHA-256 digest
A->>D: Store digest and 24-hour expiry
A-->>B: Set HttpOnly, SameSite=Lax cookie
B->>A: Authenticated request with cookie
A->>D: Find unexpired session by digest
D-->>A: Authoritative user
```

The API layers additional browser defenses around the cookie. Mutating requests with an `Origin` header are rejected unless the origin exactly matches the configured frontend URL. CORS allows that one origin and credentials rather than using a wildcard. Authentication responses are marked `no-store`. The middleware emits `nosniff`, `DENY` framing, a same-origin referrer policy, a restrictive permissions policy, and a Content Security Policy. The CSP defaults sources to self, blocks framing through `frame-ancestors 'none'`, and restricts base and form targets. Local API and media addresses are explicitly allowed for development.

CSP is mitigation, not authorization. The current policy permits inline styles and names localhost development endpoints, so deployment configuration should tighten those allowances where possible. Likewise, SameSite and origin checking reduce cross-site request forgery risk but do not excuse capability checks in the backend.

Failed login attempts are limited by the combination of client host and normalized email. Five failures within fifteen minutes produce HTTP 429 and a `Retry-After` header; success resets the entry and elapsed windows expire. This is intentionally simple and process-local. It is useful for a single local server but is not a distributed abuse-control system. A multi-instance deployment would move counters to shared storage, bound map growth, account for trusted proxy headers, and consider layered limits by IP, account, and network without letting an attacker permanently lock out one address.

Password reset without account disclosure

The forgot-password endpoint always answers with the same accepted message: “If that account exists, a reset link has been sent.” The store treats an unknown email as a successful no-op. The API therefore does not reveal whether an address is registered through status or body.

For an existing account, FrameShift generates a 32-byte random reset token, stores only its SHA-256 hash in `password_reset_tokens`, and places the raw token in the durable authentication email outbox as part of a one-hour link. Creating a new token marks all previous unused reset tokens for that user as used. Redemption hashes the presented token and requires a matching unused, unexpired row. Password replacement, token consumption, and deletion of all user sessions commit atomically.

Display Mermaid diagrams in this vault?

Only allow if you trust this vault's contents.

Allow

flowchart LR

```
R[Reset requested] --> U{Account exists?}
U -- No --> G[Return generic accepted response]
U -- Yes --> X[Invalidate older unused tokens]
X --> H[Store hash with one-hour expiry]
H --> O[Queue email containing raw link]
O --> G
L[User opens link] --> P[Hash presented token]
P --> V{Unused and unexpired match?}
V -- No --> E[Reject generically]
V -- Yes --> C[Change password, consume token, revoke sessions]
```

The raw reset secret necessarily exists in the email body and the recipient's inbox. In local development that inbox is Mailpit. The durable outbox is therefore sensitive operational data even though the reset-token table contains only hashes. Access to database backups, Mailpit, logs, and production mail infrastructure must be restricted accordingly; application code must not log message bodies or URL tokens.

Personal and organization workspaces

Account creation and personal workspace creation occur in one transaction. Every user immediately receives a `PERSONAL` workspace with themselves as owner, a default 100 GiB logical storage quota, and three active render slots. This removes the special case of “projects without a workspace”: personal and collaborative work use the same ownership, capability, quota, and project-linking machinery.

Users can also create `ORGANIZATION` workspaces. An organization is not a separate authentication realm; it is a shared authority boundary with one owner and a membership table. Projects link to a workspace through `project_workspaces` , and membership is copied into project membership where existing project queries require it. Team changes update that project-facing membership transactionally, keeping older project access paths consistent with workspace authority.

The role names are intentionally understandable to a studio team, but operational authorization resolves them to named capabilities:

Role	Create projects	Write scenes	Review scenes	Create renders	Control jobs	Manage team/settings	Transfer ownership
Owner	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Admin	Yes	Yes	Yes	Yes	Yes	Yes	No
Artist	Yes	Yes	No	Yes	Yes	No	No
Operator	No	No	No	Yes	Yes	No	No
Viewer	No	No	No	No	No	No	No

Administrative operations use role rank where hierarchy is the policy: owner and admin can update settings or manage non-owner members, while ownership transfer requires the actual owner. Operational operations ask for a specific capability such as `scene.write` , `scene.review` , `render.create` , or `job.control` . The store resolves the user's current workspace role and checks that capability at the point of mutation.

This is stronger than checking what the UI displays. A user can call the API directly, replay an old request after a role change, or run a stale frontend bundle. Even backend code that asks “is

this person an Artist?” is brittle: later, a Producer role might review and publish without writing scenes. Capability checks describe the operation's requirement and centralize the mapping from organizational policy to permission. The UI may use the same mapping to hide unavailable controls for clarity, but the store's check is the security boundary.

Invitations and the durable email boundary

Only an owner or admin can invite a member, and `OWNER` is not an assignable invitation role. The API generates a random invitation token. The authoritative invitation row stores its hash, status, target email, proposed role, inviter, and expiry. The raw token is written only into the invitation email body in `invitation_outbox`; it is not returned in workspace listings and is not stored in `workspace_invitations`.

This phrasing deserves precision: the outbox itself is durable database state so that delivery survives restarts. “Hash tokens at rest” applies to the long-lived authoritative invitation record, while the queued email must temporarily contain the deliverable secret. Mailpit then durably captures that email for local inspection. Protecting the outbox and Mailpit volume is therefore part of protecting invitation credentials.

Acceptance requires three facts at once: the hash matches a pending invitation, the invitation has not expired, and the signed-in user's email equals the invited email case-insensitively. The transaction inserts workspace and project memberships, marks the invitation accepted, and appends an audit entry. Cancellation changes only a pending invitation. The status predicate prevents canceled or already accepted tokens from being reused.

Display Mermaid diagrams in this vault?

Only allow if you trust this vault's contents.

Allow

```
sequenceDiagram
    participant M as Admin
    participant S as Store
    participant O as Invitation outbox
    participant P as Mailpit
    participant I as Invitee
    M->>S: Invite email with role
    S->>S: Store token hash and PENDING status
    S->>O: Store raw token in email body
    O->>P: SMTP delivery with retries
    I->>S: Accept raw token while signed in
    S->>S: Hash token; verify status, expiry, and email
    S->>S: Add memberships, mark ACCEPTED, append audit
```

Alternatives include returning the token to the inviter for manual sharing or using an identity provider's organization invitation system. Manual sharing makes delivery easy but creates

uncontrolled copies and weaker observability. An external identity platform can add federation and polished lifecycle management, but it introduces network and vendor dependencies into a product whose default installation is local. The current design keeps the workflow testable and authoritative inside FrameShift.

Ownership, audit, and quotas

Ownership transfer is deliberately explicit. The actor must currently be the owner, and the new owner must already be a workspace member. In one transaction, the old owner becomes an admin, the selected member becomes owner, `owner_user_id` changes, and an audit entry records the transfer. Ordinary member updates cannot assign `OWNER`, and removal refuses to delete the owner. Those constraints prevent a workspace from becoming ownerless or acquiring two owners through generic role editing.

Workspace audit entries record the actor, action, target type, target identifier, structured payload, and timestamp. Creation, settings changes, invitations, membership changes, transfers, project creation, review transitions, and notes all append records in the same transaction as the action. The activity feed is consequently an account of committed authority changes, not a best-effort log emitted after the fact. Entries are read newest first and scoped to users who are still workspace members.

Quotas are also backend decisions. Storage usage includes scene-version files, present packaged assets, and render artifacts. An upload is rejected when its additional bytes would exceed the workspace storage quota. Render admission checks both current storage use and the count of jobs in active lifecycle states against the workspace render limit. The figures returned to the UI are useful feedback, but repeating the calculation before authoritative writes prevents a modified client from bypassing limits.

The current checks are logical application enforcement, not filesystem reservations. Concurrent uploads can both observe remaining capacity before either finalizes, and SQLite serialization only protects the transactions that actually surround a check and mutation. Higher-scale deployments may need reservation rows, database constraints, or serialized quota accounting. The existing approach favors clarity and local operation while keeping the policy in the backend.

Durable notifications and Mailpit

When a render completes or terminally fails, FrameShift inserts an in-app notification and an `email_outbox` row for each project member in the same transaction as the job transition. It also emits a notification event. Job completion does not depend on SMTP availability: the durable intent to notify commits first, and a separate server loop performs delivery later.

The loop checks the general notification outbox, invitation outbox, and authentication outbox. A claim changes a due `PENDING` or `RETRY` row to `SENDING` and increments its attempt. General

and invitation claims also recover a `SENDING` lease older than five minutes, covering a process crash after claim. Successful SMTP delivery changes the row to `SENT`. Failure records the error and schedules exponential minute backoff, capped in exponent, until the fifth attempt becomes `FAILED`.

```
PENDING -> SENDING -> SENT
      |
      +--> RETRY -> SENDING
      |
      +--> FAILED after max attempts
```

This is the transactional outbox pattern. Sending SMTP inside the render transaction would hold database work open across the network and could roll back a completed render because a mail server was down. Sending after commit without an outbox creates the opposite gap: a crash between commit and send loses the notification forever. The outbox turns that gap into explicit durable work.

Delivery is at least once around crash boundaries. If SMTP accepts a message and the server crashes before marking it sent, the lease may be retried and the recipient may receive a duplicate. The stable delivery ID becomes the SMTP `Message-ID`, which can help downstream deduplication, but exactly-once email is not a realistic end-to-end guarantee. Idempotent content and operational visibility are better tools than pretending otherwise.

Mailpit's purpose is development safety and observability. Docker Compose exposes SMTP on `localhost:1025`, its web inbox on `localhost:8025`, and persists messages in a named volume. Developers can exercise registration, reset, invitation, completion, and failure email without credentials, internet access, or accidental delivery to real people. Mailpit is not the production provider; the mailer speaks ordinary SMTP, so deployment can point it at managed infrastructure with suitable transport and authentication changes.

The mailer parses sender and recipient addresses, strips CR and LF from subjects to prevent header injection, encodes the subject, normalizes body line endings, and applies connection and operation deadlines. Those details matter even in a local tool because invitation addresses and notification text cross a protocol boundary.

Review as a production gate

Every new scene version receives a review row in `DRAFT`. A member with `scene.write` can submit a draft or rejected version to `REVIEW` and must assign a user who currently has `scene.review`. Submission records who submitted, the assigned reviewer, and the time while clearing any previous decision. Only that assigned reviewer can change the item from `REVIEW`

to `APPROVED` or `REJECTED` ; merely possessing review capability is not enough to decide someone else's assignment.

Notes are a chronological conversation attached to one immutable scene version. Any workspace member can add a trimmed note between 1 and 2,000 characters. Each note records its author and time, and adding it creates both an audit entry and an event. Notes do not themselves alter review state, which preserves the distinction between discussion and an authoritative decision.

Display Mermaid diagrams in this vault?

Only allow if you trust this vault's contents.

Allow

stateDiagram-v2

```
[*] --> DRAFT: scene version created
DRAFT --> REVIEW: submit and assign reviewer
REVIEW --> APPROVED: assigned reviewer approves
REVIEW --> REJECTED: assigned reviewer rejects
REJECTED --> REVIEW: revise context and resubmit
APPROVED --> [*]
```

The gate is enforced when production work is created. A normal render query joins scene versions to a review row whose status is `APPROVED` ; if no approved version is available, job creation returns `ErrSceneApprovalRequired` . Calibration is the deliberate exception: it uses the latest scene version so artists can test representative frames before approval. Production rendering selects the newest approved version, which may be older than the latest uploaded draft. That behavior keeps an approved source renderable while a newer revision is still being discussed.

This database-level selection is stronger than asking the frontend to disable “Render.” It also records the exact approved `scene_version_id` in the job source, so later uploads do not silently change an existing job. An alternative is to approve each render request rather than each scene version. That offers tighter per-delivery control but repeats approval for identical source material. Another alternative is allowing any reviewer to decide any queued review; that reduces bottlenecks but weakens accountability. FrameShift chooses version approval with explicit assignment because it maps cleanly to a small studio's handoff.

Desktop notifications are a presentation layer

The dashboard polls authoritative in-app notifications and allows one or all to be marked read through user-scoped API operations. Desktop notifications are an optional enhancement built on the browser Notification API. Permission is requested only after the user enables it. When a newly observed notification differs from the last known ID, the page creates a desktop

notification only if permission is granted and the document is hidden. The initial fetched item establishes a baseline rather than producing a burst for old history.

This layer is intentionally best effort. Browser permission can be denied, the tab can be closed, and the operating system can suppress alerts. Read state remains in the database, email remains in its durable outbox, and render state remains authoritative in the backend. A service worker and Web Push could notify closed browsers, but they would add subscription lifecycle, key management, and another delivery service. For a local-first dashboard, in-app history plus durable email and foreground browser notifications provides useful coverage without confusing presentation delivery with system state.

The resulting boundary

Across these features, FrameShift follows one repeatable pattern. Human secrets use a deliberately expensive password hash; random bearer secrets use fast one-way hashing and limited lifetimes. Cookies carry opaque credentials, while the database controls expiry and revocation. Roles communicate team structure, while capabilities authorize operations at the store boundary. Email is durable work rather than part of an HTTP or render transaction. Review decisions are explicit state transitions, and production jobs select only approved source versions.

The tradeoff is more tables and more transitions than a prototype with a password column, a role check in React, and direct SMTP calls. The benefit is that failure has a place to land. A restarted server can retry mail. A stolen database does not reveal active session tokens. A stale browser cannot retain authority after a role change. A rejected version cannot become production input because a button was mistakenly enabled. That is the practical meaning of making the backend the system of record: collaboration remains understandable even when users, processes, and networks do not behave in the convenient order.

Durable application state still needs an operational home, a backup boundary, and a deployment procedure that respects its guarantees.

Part 9 — Operating FrameShift: Storage, Backups, PostgreSQL, Docker, and Deployment

FrameShift is local-first, but local-first does not mean operationally casual. The Go backend is the system of record, and its database coordinates files that may be large, expensive to reproduce, or temporarily owned by another process. A useful deployment therefore has to answer several questions at once: where authoritative state lives, how files age out, what a backup actually contains, how a restore avoids half-installed state, and which development conveniences must change before the service is exposed beyond one workstation.

The default installation deliberately keeps the answer simple. SQLite stores authoritative relational state, managed files live beneath one data directory, Mailpit receives development email, and the web application and worker communicate with the Go server over localhost. PostgreSQL and S3-compatible object storage are optional extensions. They improve concurrency or capacity in particular deployments, but they also add independent failure domains and backup responsibilities.

This chapter describes the behavior implemented by the server, admin command, store adapter, object-store client, Compose services, Makefile, and macOS installer. It also calls out boundaries that operators must not mistake for guarantees.

The Data Directory Is the Unit of Local Ownership

`FRAMESHIFT_DATA_DIR` defaults to `./var`. The server creates `storage/` below it and, when PostgreSQL is not configured, opens `frameshift.db` beside that directory. Backups and restore plans also sit at the data-directory level:

```
var/
├─ frameshift.db
├─ backups/
│   └─ frameshift-backup-20260907-120000-ab12cd34.zip
├─ restore-pending.json
└─ storage/
    ├─ uploads/
    ├─ blobs/
    ├─ projects/
    ├─ archive/
    └─ published/
```

Not every directory exists immediately. They are created as workflows need them. The important division is between database metadata and managed file content:

- `frameshift.db` contains users, sessions, workspaces, projects, scene versions, jobs, frame tasks, attempts, notifications, artifact lifecycle records, and other authoritative state.
- `storage/uploads` holds resumable `.part` uploads. Upload sessions expire in the database and are not finished scene content until ingestion succeeds.
- `storage/blobs` is content-addressed scene asset storage. Archive ingestion places blobs under a checksum prefix so identical packaged assets can be reused.
- `storage/projects/{projectID}/scenes/{sceneID}/versions/{version}` contains ingested scene versions and package material.
- `storage/projects/{projectID}/jobs/{jobID}/frames` and `artifacts` contain frame outputs and finalized deliverables.

- `storage/archive/{artifactID}` is the local COLD tier when no object store is configured.
- `storage/published` is the managed local publication area used by delivery workflows.
- `backups` contains promoted backup ZIP files, outside `storage` but inside the same data directory.

Database rows use relative, slash-separated storage paths. API code rejects absolute or non-local paths before destructive cleanup, and object-backed paths use a distinct `object://` prefix. This prevents a stored artifact path from casually turning cleanup into an arbitrary filesystem delete. It does not remove the need for ordinary operating-system permissions: run FrameShift as a dedicated user where practical, grant that user access only to its data and publication destinations, and do not make the managed storage root world-writable.

For the packaged macOS installation, the equivalent root is:

```
~/Library/Application Support/FrameShift/
├─ bin/
├─ web/
├─ data/
│   ├── frameshift.db
│   └── storage/
└─ logs/
```

Treat the whole `data` directory as one local recovery unit. Copying only the SQLite file loses scene packages and render artifacts; copying only `storage` loses the metadata that assigns meaning, ownership, checksums, and lifecycle state to those files.

Retention, Cleanup, Pinning, and Archive Tiers

FrameShift has two retention settings: backup days and artifact days. Both default to 30 and 90 days respectively, and the admin API accepts values from 1 through 3650. When finalization creates an artifact, it also creates an `artifact_lifecycle` row with tier `HOT`, `pinned=false`, the original relative path, and a `retain_until` timestamp calculated from the artifact retention setting at that moment.

That last detail matters. Changing `artifact_retention_days` affects newly finalized artifacts; it does not rewrite every existing `retain_until` value. Retention is therefore a policy applied at creation, not a continuously recomputed global expiration date.

The tiers mean:

- **HOT**: the artifact remains at its normal path under the job's local artifact directory.

- **COLD without object storage:** the server renames the file into `storage/archive/{artifactID}/{filename}` and updates its database path.
- **COLD with object storage:** the server uploads the file to `artifacts/{artifactID}/{filename}`, records an `object://...` path, commits the lifecycle change, and then removes the local copy.
- **Pinned:** automated cleanup excludes the artifact regardless of its `retain_until` timestamp.

Moving an artifact back to HOT uses the lifecycle row's original path. Local files are renamed back. Object-backed files are downloaded to a temporary sibling, chmodded to `0640`, and atomically renamed into place; after the database update succeeds, the old object is removed. The API has compensating actions for common failures: it renames a locally moved file back, deletes a newly uploaded object, or removes a downloaded local file if the database update fails. These are practical rollback measures, not a distributed transaction spanning SQLite or PostgreSQL and S3.

Publishing selected final artifacts pins them. This protects the video, preview, thumbnail, and manifest associated with an approved publication from normal cleanup. Pinning is a preservation flag, not an additional copy and not proof that an external destination still contains the publication.

Admin cleanup processes at most 1,000 eligible artifacts per request. It deletes unpinned artifacts whose stored `retain_until` is before the current time, then removes the artifact row after the file or object deletion succeeds. A failed object deletion, missing object-store client, unsafe local path, or unexpected filesystem error causes that candidate to be skipped. Missing local files are tolerated so stale metadata can still be removed. The same cleanup request removes local backup ZIPs whose filesystem modification time is older than the configured backup retention period.

An operator should use a repeatable cleanup routine:

1. Review pinned and published deliverables before changing retention.
2. Confirm the object-store endpoint is reachable before cleaning object-backed artifacts.
3. Run cleanup repeatedly if more than 1,000 expired artifacts may exist.
4. Compare released bytes with filesystem or bucket usage; sparse files, ZIP compression, and filesystem allocation mean the values need not match exactly.
5. Investigate skipped objects and unsafe paths instead of deleting database rows manually.
6. Keep at least one tested backup outside the data directory before aggressive cleanup.

Archiving is primarily a capacity and access-tier feature. A local COLD move does not protect against loss of the same disk. An object-store move can protect against workstation loss only

when the bucket itself has independent durability, versioning or replication, access controls, and a tested recovery procedure.

Online Backups and Their ZIP Manifest

The admin backup endpoint first checks authoritative state. It refuses to start while any render is active or any upload is in `UPLOADING` or `FINALIZING`. This pause requirement prevents the file walk from racing the most obvious producers of managed content. It is stricter than SQLite technically requires, but it improves the consistency between database references and copied files.

For SQLite, `BackupTo` executes `VACUUM INTO` to a new file in a temporary backup directory. SQLite produces a consistent online snapshot without requiring the server process to close its live database connection. The server then writes a ZIP containing:

```
frameshift.db
storage/...
manifest.json
```

For PostgreSQL, the database member is `frameshift.dump`, a custom-format `pg_dump`, while managed local files still appear under `storage/`.

The version 1 manifest records the creation time, database filename, database backend, storage prefix, and administrative statistics captured before the backup. A representative shape is:

```
{
  "formatVersion": 1,
  "createdAt": "2026-09-07T12:00:00Z",
  "database": "frameshift.db",
  "databaseBackend": "sqlite",
  "storage": "storage/",
  "statistics": {
    "users": 3,
    "projects": 18,
    "jobs": 72,
    "activeJobs": 0,
    "activeUploads": 0
  }
}
```

The archive is built at mode `0600` in a temporary directory and promoted into `backups/` with a rename only after the ZIP closes successfully. Restore validation rejects empty archives, more

than 100,000 entries, path traversal, absolute paths, backslashes, symbolic links, expansion beyond 1 TiB, unsupported format versions, unexpected database names, and an incompatible storage prefix. Extraction creates files exclusively at mode `0600` and only extracts the recognized database member and `storage/` tree.

Those checks make the ZIP structurally defensive, but “secure manifest” must not be read as “cryptographically authenticated manifest.” Version 1 does not sign the manifest or list a digest for every backup member. ZIP CRCs detect accidental corruption but are not an authenticity mechanism. Store exported backups on authenticated media, restrict read access because they contain account and project data, encrypt them at rest where required, and verify provenance before restore.

There is another important boundary: a backup walks local `storage/`. If a COLD artifact has moved to `object://`, its bytes are no longer local and are not copied into the ZIP. Back up the object bucket independently and coordinate its retention with FrameShift backup retention. A database restore that references deleted bucket objects cannot recreate them.

Offline Restore, Atomic Promotion, and Rollback

Restore is intentionally not an HTTP operation. The admin console creates `restore-pending.json` with the archive path, backup timestamp, data directory, generated command, and `requiresShutdown=true`. An operator must then stop the backend and workers and run the command offline:

```
./bin/frameshift-admin restore \  
  --archive ./var/backups/frameshift-backup-20260907-120000-ab12cd34.zip \  
  --data-dir ./var
```

Before doing any work, `frameshift-admin` requests `${FRAMESHIFT_SERVER_URL}/healthz`. A successful response causes a hard refusal. This is a useful guard against restoring under a live server, but it is not process discovery: a server bound elsewhere, blocked by a firewall, or unhealthy at that URL may still be running. Stop and verify all server and worker processes explicitly. Workers must be stopped because they can still hold Blender processes or upload results even after the backend goes away.

The restore has a ten-minute command timeout. It validates the ZIP before extraction, expands into a temporary sibling directory, and opens the staged SQLite database so migrations and basic database validity are checked before promotion. It then renames the existing data directory to a timestamped path such as `var.pre-restore-20260907-121500`, creates a fresh data directory, and renames the staged `frameshift.db` and `storage` into it. Because staging and destination are siblings, a normal same-filesystem rename provides atomic namespace

changes. If promotion fails, restore removes the incomplete destination and renames the pre-restore directory back.

After a successful SQLite restore:

1. Keep the `*.pre-restore-*` directory until validation and a representative render complete.
2. Start only the backend and check `/healthz` , `/readyz` , and an authenticated project listing.
3. Verify scene downloads, one old artifact, user sign-in, and storage totals.
4. Start one worker and run a small render through finalization.
5. Start the remaining workers, then remove the pre-restore directory only after the rollback window closes.

Do not overstate the atomic guarantee. It depends on source, staging, and destination remaining on the same filesystem, and it does not include external publication destinations or object storage. PostgreSQL restoration is less atomic still: `pg_restore --clean --if-exists` mutates the configured database before the local storage directory is promoted. If local promotion later fails, FrameShift can roll back the local directory but cannot automatically roll the PostgreSQL database back to its prior contents. For a serious PostgreSQL restore, first snapshot or clone the database, restore into a new database where possible, validate it, and switch the connection string during a controlled maintenance window.

PostgreSQL as an Optional Authoritative Database

SQLite remains the smallest and easiest deployment. PostgreSQL is selected only when `FRAMESHIFT_DATABASE_URL` starts with `postgres://` or `postgresql://` ; otherwise FrameShift opens the local SQLite path. The Go store keeps most application queries in one SQL form through a narrow dialect wrapper around `database/sql` and transactions.

Application SQL uses `?` placeholders. For PostgreSQL, the wrapper rewrites placeholders outside SQL string literals to `$1` , `$2` , and so on. It preserves question marks inside quoted strings and understands doubled single quotes. Schema adaptation also maps the limited SQLite vocabulary used by migrations: `INTEGER PRIMARY KEY AUTOINCREMENT` becomes `BIGSERIAL PRIMARY KEY` , other `INTEGER` tokens become `BIGINT` , and `DATETIME` becomes `TIMESTAMPTZ` .

This is intentionally a small compatibility layer, not a general SQL parser. New DDL and queries must be reviewed against both engines. In particular, careless textual type replacement, engine-specific functions, locking semantics, and changes involving quoted identifiers can escape the assumptions of the adapter.

Unit tests cover placeholder rebinding and DDL conversion. Live PostgreSQL tests are opt-in through `FRAMESHIFT_TEST_POSTGRES_URL` ; they run migrations and exercise

account/workspace creation, scene reviews, scheduling, frame completion, finalization, artifact lifecycle defaults, publication claims, and a real custom-format backup. The ordinary `make test` run skips those live tests when the variable is absent. A PostgreSQL release check should therefore include:

```
docker compose --profile postgres up -d postgres
export FRAMESHIFT_TEST_POSTGRES_URL='postgres://frameshift:local-frameshift-
password@localhost:5432/frameshift?sslmode=disable'
make test
```

PostgreSQL backups prefer a host `pg_dump` found on `PATH`:

```
pg_dump --format=custom --no-owner --file <destination> <database-url>
```

If it is unavailable, FrameShift runs `docker compose exec -T postgres pg_dump` and streams stdout to a newly created mode-`0600` destination. Restore similarly prefers host `pg_restore` and falls back to the Compose `postgres` service with the dump on stdin. The Docker fallback assumes the current working directory resolves the correct Compose project, the service is named `postgres`, and the database user and name can be parsed from the URL. It is a local operational convenience, not a universal managed-PostgreSQL backup system.

For production PostgreSQL, use provider-native point-in-time recovery in addition to FrameShift ZIP backups. Match client tool versions to the server, test restores, protect connection strings, require TLS in the database URL, and monitor connection capacity. The Compose password and `sslmode=disable` examples are development-only.

Optional MinIO and S3-Compatible Cold Storage

The object-store client is disabled unless endpoint, access key, secret key, and bucket are all present. When enabled, server startup creates a MinIO-compatible client, checks for the bucket, and creates it if absent. `FRAMESHIFT_OBJECT_STORE_SECURE=true` enables TLS for that connection.

A local MinIO session can be started with:

```
docker compose --profile object-storage up -d minio
export FRAMESHIFT_OBJECT_STORE_ENDPOINT=localhost:9000
export FRAMESHIFT_OBJECT_STORE_ACCESS_KEY=frameshift
export FRAMESHIFT_OBJECT_STORE_SECRET_KEY=local-frameshift-object-secret
export FRAMESHIFT_OBJECT_STORE_BUCKET=frameshift-artifacts
export FRAMESHIFT_OBJECT_STORE_SECURE=false
```

MinIO exposes its S3 API on port 9000 and console on 9001. Its named Docker volume persists objects across container recreation, but a named volume on the same host is not an off-site backup. In a shared or hosted deployment, use a least-privilege service credential scoped to the artifact bucket, private networking or TLS, bucket lifecycle rules that do not delete live `object://` references, and independent bucket replication or backup.

FrameShift does not currently checksum an object again after upload or download against the artifact checksum, version objects, encrypt objects itself, or reconcile orphaned keys. Failed cross-system operations use compensating deletion where possible. Operators should monitor object errors and periodically compare database object paths with bucket inventory.

Docker Compose Roles and Profiles

Compose supplies infrastructure, not the FrameShift application processes. `docker compose up -d` starts Mailpit only because it has no profile. PostgreSQL and MinIO are opt-in:

```
# Development email only
make infra-up

# Add PostgreSQL
docker compose --profile postgres up -d postgres

# Add S3-compatible storage
docker compose --profile object-storage up -d minio

# Start all three
docker compose --profile postgres --profile object-storage up -d
```

Mailpit listens for SMTP on 1025 and exposes its inbox UI on 8025. It accepts insecure, unauthenticated development mail and persists messages in `mailpit-data`; never expose it as a production SMTP relay. PostgreSQL 17 listens on 5432, persists to `postgres-data`, and uses `pg_isready` for its healthcheck. MinIO persists to `minio-data` and has a readiness healthcheck. The configured credentials are visible development defaults and must not cross a trust boundary.

`docker compose down` stops the services but leaves named volumes. Deleting volumes is a destructive operation outside FrameShift's backup workflow. Before doing so, export any PostgreSQL or MinIO state required for recovery.

Environment and Transport Security

`.env.example` is a configuration inventory, not a secret store. Copying it to `.env` is useful for the Next.js and Compose tools that load environment files, but the Go binaries do not include a

dotenv loader. Variables must be exported by the invoking shell, process supervisor, container runtime, or LaunchAgent environment.

At minimum, replace the development worker token and webhook secret, set public frontend and API URLs consistently, and decide whether TLS terminates in FrameShift or at a reverse proxy. Native TLS requires both variables together:

```
export FRAMESHIFT_TLS_CERT=/etc/frameshift/tls/fullchain.pem
export FRAMESHIFT_TLS_KEY=/etc/frameshift/tls/private-key.pem
export FRAMESHIFT_SECURE_COOKIES=true
export FRAMESHIFT_FRONTEND_URL=https://frameshift.example.com
```

If only one TLS path is set, the server refuses to start. `FRAMESHIFT_SECURE_COOKIES=true` marks the HTTP-only, `SameSite=Lax` session cookie Secure; enable it whenever browsers use HTTPS. When TLS terminates at a reverse proxy, the backend may still speak HTTP internally, but the browser-facing origin and cookie setting must describe the HTTPS deployment.

CORS allows one configured `FRAMESHIFT_FRONTEND_URL`, credentials, and a fixed set of methods and headers. State-changing requests with a different nonempty `Origin` are rejected. This is not a multi-origin allowlist. The server also emits content-type sniffing, frame, referrer, content security, permissions, and auth no-store headers. The current content security policy contains localhost API endpoints, so deployments on different public origins should verify browser behavior and may need a code-level policy update or same-origin reverse-proxy routing.

Never expose default worker credentials, PostgreSQL, MinIO, Mailpit, metrics, or pprof directly to an untrusted network. Firewall infrastructure ports and rotate any credential that has appeared in shell history or a checked-in environment file.

Metrics, Profiling, and Safety Limits

`GET /metrics` emits Prometheus text for total and active jobs, active uploads, worker online state, and active leases. It is protected by the same bearer token used by workers, so a scraper must send `Authorization: Bearer <FRAMESHIFT_WORKER_TOKEN>`. The token is powerful worker authentication, not a metrics-only credential; place the scraper on a trusted network and avoid broad token distribution.

Pprof is opt-in through `FRAMESHIFT_PPROF_ADDR`. The server accepts only an address whose host is an explicit loopback IP, for example:

```
export FRAMESHIFT_PPROF_ADDR=127.0.0.1:6060
```

Values such as `:6060`, `localhost:6060`, or a non-loopback address are refused. Access remote profiling through a controlled SSH tunnel rather than widening the listener. Pprof can expose memory contents, request details, and performance-sensitive internals.

The system endpoint samples host CPU, virtual memory, and filesystem usage for the managed storage path. Defaults block new scheduling at 99% CPU, 90% memory, 95% disk usage, or less than 10 GiB free. Disk usage at 90% or above is a warning even before the blocking percentage is reached. Configure them with:

```
FRAMESHIFT_MAX_CPU_PERCENT
FRAMESHIFT_MAX_MEMORY_PERCENT
FRAMESHIFT_MAX_DISK_PERCENT
FRAMESHIFT_MIN_DISK_FREE_BYTES
```

Both disk conditions matter. A large volume can have more than 10 GiB free while already exceeding the percentage limit; a small volume can be below the free-byte floor before reaching 95%. Thresholds prevent claims when telemetry says the host is unsafe, but they are not disk reservations. Concurrent workers, finalization, backups, uploads, and unrelated processes can consume space after a check. Capacity planning should include peak frame output, temporary finalization files, one full local backup, restore staging, and a pre-restore copy at the same time.

GPU telemetry uses `nvidia-smi` when available. On macOS, FrameShift reports Apple GPU utilization as unavailable rather than invoking privileged `powermetrics`. Worker heartbeats still report process CPU and RSS memory, and completed attempts retain metrics for job profiles.

Makefile, LaunchAgents, Readiness, and Doctor

The Makefile is the normal developer entry point:

```
make setup      install Bun and Go dependencies; create var/storage
make infra-up   start default Compose services (Mailpit)
make server     run the Go backend
make worker     run a local worker
make web        run the Next.js development server
make test       run Go tests, frontend lint, and frontend production build
make e2e        run Playwright tests
make build      build three Go binaries and the web application
make doctor     check local executables and services
make install    build and install macOS LaunchAgents
make uninstall  unload agents while preserving data
```

`make install` copies server, worker, and admin binaries under `~/Library/Application Support/FrameShift/bin`, synchronizes the web source, performs a frozen Bun install and

production build, and writes three per-user LaunchAgents. The agents run at login, remain alive, share fixed localhost URLs, write logs under the installation directory, and use the packaged `data` directory. The web agent records the absolute Bun executable found during installation.

The installer embeds only a small set of environment values: data directory, local URLs, public API URL, Blender path. It does not import `.env`, object-store credentials, PostgreSQL URL, TLS settings, custom worker tokens, SMTP overrides, or safety tuning. Operators needing those settings must extend the LaunchAgent environment securely or use another supervisor. Re-running installation uses `rsync --delete` for the packaged web tree but does not erase data. `make uninstall` unloads and removes the three plist files; it intentionally preserves application data, binaries, web files, and logs.

`/healthz` and `/readyz` currently return the same process-level success payload. Readiness does not probe the database, object store, SMTP server, Blender, FFmpeg, free disk, or workers. Use it to determine that the HTTP handler is responding, not that every dependency is usable.

`make doctor` checks Blender, FFmpeg, backend `/healthz`, and the Mailpit information endpoint with short timeouts. It prints `OK` or `MISSING` and exits nonzero if any check fails. PostgreSQL, MinIO, TLS certificates, storage permissions, disk capacity, frontend availability, and worker connectivity are not currently doctor checks. Optional or intentionally stopped local services can therefore make doctor red, while omitted production dependencies can still leave it green.

Deployment Checklist

Before first production or studio-network use:

1. Choose SQLite or PostgreSQL explicitly and test the exact backup and restore path.
2. Put `FRAMESHIFT_DATA_DIR` on a filesystem with enough peak and restore headroom.
3. Back up the data directory and any object bucket independently off-host.
4. Replace worker, webhook, database, object-store, and SMTP development credentials.
5. Enable HTTPS, secure cookies, and a frontend origin matching the browser-visible URL.
6. Restrict backend, metrics, infrastructure, and profiling listeners at the network layer.
7. Set retention and safety thresholds from measured render sizes and host capacity.
8. Run live PostgreSQL tests when PostgreSQL is selected.
9. Verify `/healthz`, `/readyz`, authenticated API access, one upload, one render, finalization, email, and artifact download.
10. Perform a restore rehearsal onto disposable storage and record its duration and required free space.

For routine operations:

1. Monitor disk free bytes as well as percentage, active leases, failed jobs, and backup age.
2. Review cleanup output and rerun it when the 1,000-candidate batch can be exceeded.
3. Test an exported ZIP and the corresponding object-store recovery set regularly.
4. Keep pre-restore directories only for a defined rollback period, then remove them deliberately.
5. Apply upgrades first to a recoverable copy and run `make test`, including configured integration tests.

Tradeoffs and Remaining Constraints

FrameShift's operational model favors inspectable local state and small deployment machinery. SQLite plus managed files is easy to understand, copy, and restore. Online snapshots avoid database downtime during backup, staging avoids exposing partial archives, and offline promotion gives SQLite deployments a practical rollback path. PostgreSQL and object storage can be adopted independently when workload or capacity warrants them.

The cost of that simplicity is that several guarantees remain procedural. The backup manifest is validated but unsigned. Backups do not contain object-backed COLD bytes. Cleanup is manually invoked and bounded per request. Readiness is process-level only. Doctor covers a development-oriented subset. The SQL dialect adapter is deliberately narrow. PostgreSQL restore cannot atomically coordinate database replacement with local file promotion, and object-store lifecycle changes rely on compensation rather than distributed transactions. LaunchAgents provide a convenient single-user macOS package, not secret management, sandboxing, rolling deployment, or multi-host orchestration.

Those are acceptable constraints when they are visible. A reliable operator closes the gaps with restricted service accounts, external monitoring, off-host and bucket backups, tested maintenance procedures, capacity margins, and restore rehearsals. The central rule remains the same as elsewhere in FrameShift: preserve the backend's authoritative state and the managed files it references as one coherent operational system.

The final part brings those backend guarantees into the interface and examines the tests and lessons that keep the product honest.

Part 10 — Turning Infrastructure into a Product: UI, Testing, and Lessons Learned

FrameShift began with infrastructure questions: how to inspect an untrusted Blender file, represent a render as durable work, lease frames to workers, recover abandoned attempts, and finalize trustworthy artifacts. Those mechanisms are necessary, but users do not experience a

lease table or a transaction. They experience whether the product tells them what is happening, lets them make the next safe decision, and remains understandable when something fails.

The web application is where those backend guarantees become visible. It does not replace the Go backend's authority. It translates durable state into an operational workspace and sends user intent back as explicit commands. That distinction shaped the routes, dashboard information architecture, live console, recovery controls, scene review workflow, output delivery, accessibility work, and the testing strategy around them.

This final part explains that product layer and the engineering lessons behind it. It also records failures found while building the whole system. Some were ordinary implementation bugs. Others exposed incorrect assumptions about Git, Blender files, process working directories, worker identity, or database time values. All were useful because they sharpened the boundaries described throughout this series.

From Routes to a Product Journey

The Next.js application has a deliberately small public route surface:

- `/` introduces FrameShift as local Blender render operations rather than a generic cloud service.
- `/login` and `/register` provide focused account entry points.
- `/forgot-password` starts local password recovery.
- `/reset-password?token=...` consumes the reset token and invalidates existing sessions after a successful change.
- `/dashboard` contains the authenticated operational product.

The landing page is not a substitute dashboard. Its full-bleed dashboard image, short capability list, and local-first signal explain the actual offer: queue, observe, recover, review, and deliver renders using the Blender installation and storage under the operator's control. Its calls to action lead to account creation, sign-in, or the dashboard. The copy also introduces the system's real differentiators, including immutable review, recoverable storage, worker-aware scheduling, and checksummed delivery.

Authentication routes stay visually and behaviorally narrow. Registration asks for name, email, and a password of at least ten characters, then creates a personal workspace through the backend. Login uses the same component in a different mode and sends credentials with the request so the server-managed session cookie remains authoritative. Errors are rendered beside the form, and successful authentication replaces browser history with `/dashboard` rather than leaving a credential form in the back stack.

Password recovery is intentionally non-enumerating from a user's perspective: the success message says that a link is sent "if the account exists." In local development, Mailpit makes that

email observable without requiring an external provider. The reset route reads the token from the query string, disables submission when it is missing, and explains the important security consequence: resetting the password signs out all existing sessions.

The dashboard still checks `/api/v1/auth/me` before showing operational data. If a user visits it without a valid session, an embedded authentication gateway is shown. That duplication with the dedicated login route is an honest tradeoff. It makes a direct dashboard visit recover gracefully, but it also means there are two authentication presentations to keep behaviorally aligned. A future route guard and shared shell could reduce that duplication without moving authorization into the frontend.

Why Next App Router, Tailwind, and Custom CSS

Next.js App Router fits this product because route ownership, metadata, font loading, images, and client-only operational surfaces can coexist without introducing a separate routing package. The root layout defines the language, metadata, Instrument Sans, and JetBrains Mono once. Public pages can remain simple server-rendered components, while forms and the dashboard opt into client behavior with `"use client"` where browser state, navigation, EventSource, notifications, and polling are actually needed.

Tailwind CSS 4 is included through a single import:

```
@import "tailwindcss";
```

Only a small number of utilities appear in the JSX. Most product styling lives in `globals.css` as named classes and a compact token set:

```
:root {
  --canvas: #f4f3ef;
  --paper: #fff;
  --ink: #20211e;
  --muted: #5f615c;
  --nav: #20231f;
  --coral: #c9472d;
  --green: #247054;
  --blue: #4275a7;
  --gold: #aa7420;
}
```

This hybrid is useful during product discovery. Tailwind provides a modern build pipeline and occasional layout utilities, while custom CSS gives dense operational components stable names such as `live-console`, `queue-table`, `version-rail`, and `admin-worker-list`.

Those names make the stylesheet searchable by product concept, and CSS variables keep status colors and surfaces consistent.

There are costs. A large global stylesheet has weak component isolation, and the dashboard route is a large client component containing many drawers and workflows. Tailwind is currently underused relative to its dependency cost. The sensible next step is not a visual rewrite. It is to extract dashboard features into components with colocated tests while preserving tokens and behavior. CSS Modules or cascade layers could then create ownership boundaries without replacing the established visual language.

The interface uses a restrained light canvas, dark navigation, white work surfaces, and several semantic accent families instead of one dominant hue. Instrument Sans supports compact labels without looking like a default browser UI, while JetBrains Mono makes IDs, timestamps, progress, and measurements easy to distinguish. Lucide icons provide familiar action symbols, with text retained where the consequence needs to be explicit.

Information Architecture: Scan First, Drill Down Second

The fixed desktop sidebar establishes five primary pages: Overview, Projects, Renders, Scenes, and Outputs. They are views within the dashboard rather than independent URL routes today. This keeps workspace state and live subscriptions stable while an operator moves between tasks, although it means deep links and browser back/forward history do not yet identify a selected dashboard page.

Overview answers the first operational questions: how many renders are active, how many frames are queued, how many jobs completed, and how much validated output exists. It places the active render and system health beside each other, followed by the latest output and queue. This is a monitoring view, not a decorative analytics page.

Projects and Scenes organize inputs. Projects group production work, while Scenes expose inspected files and immutable versions. Renders is the lifecycle view, with counts and filters for active, completed, failed, or canceled work. Outputs begins at the other end of the pipeline: finalized artifacts ready to inspect, retain, archive, download, or publish.

The second sidebar section is workspace-scoped. The switcher displays workspace type and the current role, then exposes workspace creation, members, and settings. The role remains visible because it explains why an action may be disabled. Viewers can inspect but cannot start renders; owners and administrators receive broader team, review, destination, and maintenance controls.

Profile and machine administration are intentionally separate. The profile menu owns personal identity, account security, session revocation, workspace settings, and sign-out. The Admin entry appears only after the backend confirms machine-administrator access. Its drawer owns

installation-wide statistics, worker fleet controls, retention, cleanup, backups, and restore planning. A workspace owner is not automatically the machine administrator, and the navigation should not imply otherwise.

This architecture follows a scan-first, drill-down-second rule. Tables and summaries show enough information to choose an item. Drawers then hold focused detail without discarding the surrounding workspace. The job inspector, scene review, account security, workspace manager, new-render dialog, and admin console all use dialog semantics and labeled headings. The drawback is drawer depth: opening too many related tasks can become spatially cramped, and current dialogs do not yet expose URL-addressable state.

Responsive Desktop and Mobile Behavior

Desktop layout prioritizes repeated operational use. A 236-pixel fixed sidebar keeps major destinations stable, the top bar keeps search and identity available, and the main area can display metric grids, active render detail, system status, queue rows, and output delivery together. At widths below 1,100 pixels, four metrics become two columns, the active-render and system panels stack, and output preview and delivery become one column.

At 800 pixels, the sidebar transforms off-canvas and is opened with a labeled menu button. The app shell loses its desktop offset, profile detail contracts to an avatar, keyboard-hint decoration disappears, and queue tables become horizontally scrollable rather than compressing operational columns into unreadable fragments. At 620 pixels, page headings, metric cards, render-stage columns, resource rows, and administrative forms stack. At 480 pixels, lifecycle counts become a single column.

Scene review has a purpose-built mobile adaptation at 700 pixels. The vertical version rail becomes a horizontal strip, its active marker moves from the left edge to the bottom edge, the thumbnail and metadata stack, and the four comparison facts become a two-by-two grid. This is more useful than merely shrinking the desktop drawer because version selection remains immediately available.

Responsive design still has limits. A wide queue is scrollable on mobile, not transformed into cards, because status, progress, ETA, and actions benefit from column alignment. That choice favors information fidelity over one-handed convenience. Touch-target and real-device testing should continue, especially for nested drawer controls, long filenames, virtual keyboards, and landscape phones.

Motion is used to communicate active work: progress sweeps, status pulses, and live indicators make a running render visually distinct from a static record. `prefers-reduced-motion: reduce` disables those animations and leaves a stable progress indication. Motion is therefore supplementary, never the only carrier of state.

Live Console, Worker Control, and Recovery

The live render experience is where infrastructure most visibly becomes a product. The overview combines frame number, phase, percentage, elapsed time, estimated remaining time, confidence, attempts, CPU, memory, and worker health. Opening the job inspector reveals tabs for live state, frames, report, and logs.

The live tab polls frame, report, and projected status every 1.5 seconds while the job is active. Logs refresh every second and can be filtered or downloaded. The page also subscribes to server-sent events for broad job and notification changes. This combination is pragmatic: events prompt responsive top-level refreshes, while polling repairs missed events and updates continuously changing progress. It creates extra requests, but it behaves better through transient local disconnects than a fragile event-only UI.

Most importantly, the frontend sends commands rather than deciding outcomes:

```
await fetch(
  `${apiURL}/api/v1/jobs/${job.id}/frames/${frame.frameNumber}/command`,
  {
    method: "POST",
    credentials: "include",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ command }),
  },
);
```

After a successful command response, the inspector refreshes. It does not directly mark a frame canceled or reset its attempt count. Pause, resume, cancel, retry, and whole-job requeue must pass backend lifecycle checks. This preserves the central rule from the architecture and scheduling chapters: the browser communicates intent; the Go store commits truth.

Recovery is made legible rather than hidden. A disconnected worker can become stale or offline while its expired frame returns to `QUEUED` with a `RECOVERING` phase. The console shows the recovery message and attempt history. Failed or canceled frames expose `Retry`, while a failed or canceled job exposes `Requeue job`. Completed frames survive a job requeue, so an operator does not pay to regenerate valid work.

The Admin fleet view adds machine-level controls. Drain prevents a worker from claiming new work while allowing active leases to settle. Resume restores eligibility. Remove is disabled while active leases exist. These controls match backend capacity tests and avoid presenting process termination as an instantaneous database toggle.

Scene Review as a Production Gate

A render manager needs more than file upload. It needs a way to answer which exact scene was approved, by whom, and what changed. The Scenes view therefore displays inspected metadata, thumbnails, project ownership, file size, latest immutable version, and review status.

The scene review drawer keeps versions in a dedicated rail. Selecting a version loads its thumbnail, filename, render engine, resolution, frame range, assigned reviewer, status, notes, and comparison with the preceding version. Comparison summarizes changed metadata and added, changed, or removed assets. This makes a new version inspectable without pretending that a four-number summary replaces artistic review.

Roles constrain the workflow. Owners, administrators, and artists can submit draft or rejected versions. The submitter selects an owner or administrator as reviewer. Only the assigned reviewer with an owner or administrator role can approve or reject. Notes remain attached to the immutable version. Production render creation can then require approval rather than relying on a mutable project-level flag.

The tradeoff is that the current review preview is a generated still, not an interactive Blender viewport or image-difference tool. It verifies provenance and supports operational approval, but it cannot expose every animation, compositing, simulation, or color-management issue. Future review should add larger frame samples, image comparison, annotations, and explicit approval of render-setting overrides without ever enabling uploaded Blender scripts in the browser or server inspection path.

Outputs, Lifecycle, and Publishing

Finalized output is presented as a delivery set, not a single download link. The latest completed job can expose a video preview, thumbnail poster, manifest, report, individual frames, and other artifacts. Each row shows type, size, and storage tier. Operators can pin an artifact, move it between hot and cold lifecycle states, or download it. The first 24 individual frame links remain directly available, with a clear note when the sequence is longer.

Publishing is a separate approval-gated workflow. Owners and administrators can configure a local-folder or webhook destination. A user requests publication to a destination, an authorized operator approves it, and the UI polls active request states such as requested, approved, delivering, or retry. The backend delivery loop performs the external side effect and records failure or completion.

That separation matters. A completed artifact is evidence that rendering and finalization succeeded. It does not prove that a webhook received the payload or a client folder accepted a copy. Publishing has its own durable request and status so external delivery cannot silently redefine render completion. The UI reflects this boundary with distinct controls and status history.

Current output presentation selects the first completed job rather than providing a full asset browser across jobs. Destination forms are compact and expose backend errors only minimally. A stronger roadmap includes job selection, artifact search, lifecycle confirmation, retry details, webhook test delivery, checksum display and verification, and pagination for large frame sequences.

Accessibility Is an Operational Requirement

Accessibility work begins with semantics and contrast, not a final audit. The palette uses dark text on light work surfaces and light text on the dark sidebar, with muted values chosen to remain readable. Status colors are paired with words such as `RUNNING`, `FAILED`, `Ready`, or `Approved`; color is never the only explanation.

A keyboard user can bypass repeated navigation through the skip link:

```
<a className="skip-link" href="#main-content">
  Skip to main content
</a>
```

Primary navigation, notification center, version rail, render summary, queue, and dialogs have accessible names. Icon-only controls receive action-specific `aria-label` values such as `Retry frame 1`, `Download manifest`, or `Close scene review`. Menus expose `aria-expanded`; drawers use `role="dialog"`, `aria-modal="true"`, and `aria-labelledby`; tabs use `role="tablist"`, `role="tab"`, and `aria-selected`. The live console uses `aria-live="polite"` so updates can be announced without interrupting the user's current task.

Desktop notifications are opt-in. The notification center asks for permission only after the operator chooses `Enable desktop alerts`. Once granted, a newly observed notification creates a browser notification only while the document is hidden. This avoids duplicating visible in-app activity and respects the browser's permission model. In-app notifications still support unread counts, mark-one-read, and mark-all-read without desktop permission.

There is more to do. Automated analysis cannot prove sensible focus order, complete keyboard operation, useful announcements under frequent polling, or compatibility with every screen reader. Drawers should gain explicit focus trapping and focus restoration if not already supplied by a future dialog primitive. Live updates may need batching so a busy render does not become noisy. Contrast should be continuously checked when tokens change, including disabled controls and small status text.

Playwright: Mock the Journey, Not the Implementation

The Playwright suite runs Chromium in desktop and iPhone 13-sized projects. It starts or reuses the Next development server, retains traces on failure, captures failure screenshots, and

retries once. API calls are intercepted at `http://localhost:8080`, allowing the browser to exercise real React rendering and interaction while the test controls backend state precisely.

The recovery mock is stateful. Initially it returns an online busy worker, a running job, a rendering frame at 35 percent, projected time, and a Blender startup log. The test opens the live render and asserts the visible progress and log. It then flips a `recovered` closure. Subsequent API responses report an offline worker and a queued frame with `RECOVERING` progress and the message "Worker disconnected; frame returned to queue." The page must update within five seconds.

This tests the user journey across several UI mechanisms without sleeping through a real Blender render:

```
let recovered = false;
await mockDashboard(page, { recovered: () => recovered });
await page.goto("/dashboard");
await page.getByRole("button", { name: /Open live render/ }).click();
await expect(page.getByText(/Rendering frame #1 . 35%/)).toBeVisible();
recovered = true;
await expect(
  page.getByText("Worker disconnected; frame returned to queue"),
).toBeVisible({ timeout: 5000 });
```

A second recovery test supplies a failed frame at attempt three, opens Renders, inspects the job, clicks Retry frame 1, and verifies that the action remains available after the mocked command. Route coverage verifies the landing, login, and registration headings. Axe runs against both the landing page and a mocked authenticated dashboard; the build fails if serious or critical violations are returned.

Mocking provides speed, determinism, and exact failure-state control. It does not prove that the Go API emits identical JSON, that cookies work across a deployed origin, that EventSource reconnects against a real server, or that Blender output reaches the UI. Contract fixtures can drift. The next layer should add a small number of full-stack Playwright journeys against an ephemeral Go server and SQLite database, while retaining network mocks for rare recovery states that are expensive to induce reliably.

The Test Pyramid Behind the Interface

FrameShift's tests are strongest when each layer proves the contract it owns. The shape is a pyramid in volume, but it is also a chain of evidence:

Display Mermaid diagrams in this vault?

Only allow if you trust this vault's contents.

Allow

flowchart TB

E2E[Playwright journeys: routes, recovery, accessibility]

INT[Integration: HTTP, PostgreSQL, MinIO, backup and publishing]

UNIT[Go unit and store tests: parsing, safety, state transitions]

UNIT --> INT --> E2E

The broad base is Go unit and local integration testing. Scene tests cover atomic storage, sanitized display names, upload bounds, invalid headers, compressed gzip and Zstandard containers, archive traversal, symlinks, excessive entries, and cleanup after failed inspection. Renderer tests cover argument construction, output validation, bounded logs, and Blender log severity. Store tests cover authorization, workspaces, sessions, notifications, reviews, upload offsets, scene versions, scheduler transitions, attempts, cancellation, retries, recovery, finalization, retention, backup, and publishing.

Scheduler tests are especially important because UI recovery language is only credible when the durable behavior is proven. `TestRecoverStalledWorkRequeuesExpiredLease` claims a frame with an already expired lease, runs recovery, checks that the frame is queued with phase `RECOVERING`, and verifies that the abandoned attempt is `FAILED`. Worker tests prove heartbeat-derived stale/offline health, drain and resume behavior, slot limits, and device compatibility.

HTTP tests exercise permissions and transport boundaries. They verify admin restrictions, session and password-reset security, resumable uploads, review actions, worker authentication, and publication behavior. These tests are faster and more diagnostic than discovering every backend regression through a browser.

PostgreSQL and MinIO are opt-in integration layers. `FRAMESHIFT_TEST_POSTGRES_URL` enables migration plus a lifecycle that creates identity and workspace data, reviews a scene, queues and claims a frame, completes finalization, reads the artifact, creates and approves a publishing request, and claims delivery. A separate test confirms `pg_dump` produces a nonempty backup. `FRAMESHIFT_TEST_OBJECT_ENDPOINT` enables a MinIO-compatible round trip that puts a file, opens and validates it, and removes it under a bounded context.

These tests skip when their environment variables are absent. That is appropriate for a dependency-light default `make test`, but it means a green local run does not automatically certify PostgreSQL or object storage. CI should have an explicit service-backed job that starts the Compose profiles and treats skips as failures for that job.

Incremental Validation as a Development Workflow

The most effective validation loop was incremental:

1. Make the smallest change that expresses one behavior.

2. Run the nearest unit or component-level check.
3. Run the package test that owns the state transition or parser.
4. Run `make test`, which executes Go tests, frontend lint, and the production Next build.
5. Run Playwright for browser behavior, responsive projects, and axe.
6. Exercise opt-in PostgreSQL, MinIO, and real Blender paths when the change crosses those boundaries.

This order makes failures legible. A blend-header parser should first fail a scene test, not a ten-minute browser journey. A responsive selector should first fail lint or build, then be seen in mobile Playwright. A scheduler change should prove attempt history in SQLite before it is trusted in the UI, then repeat its supported lifecycle against PostgreSQL.

Production build validation catches a different class of problem than development mode: server/client boundaries, route compilation, font and image handling, and TypeScript assumptions. Playwright adds interaction and rendering. Optional service tests add dialect and protocol evidence. No single command replaces the others.

The workflow also protects scope. When a failing check reveals an unrelated issue, it should be recorded rather than folded into an opportunistic refactor. Focused changes, focused checks, then the full gate made it easier to tell whether a bug came from the current hypothesis or from an existing boundary.

Bugs That Changed the Design

Several discoveries deserve to be preserved because each represents a reusable engineering lesson.

The stale Gitlink

An early frontend scaffold retained nested-repository metadata, so the parent repository treated `apps/web` as a gitlink rather than ordinary files. Changes inside the app could appear missing or remain tied to a stale submodule-like entry. The correction was to remove the nested Git metadata and stage the actual tree. The lesson is to inspect `git status`, `git ls-files --stage`, and repository boundaries immediately after using a project generator. Source can compile perfectly while version control records the wrong object.

Blender version formats

Blender version output is human-facing and not guaranteed to be one exact token shape. Depending on invocation and build, the first line may include `Blender 5.1`, a patch version, build metadata, or surrounding output. Treating it as a rigid semantic-version field caused detection and display assumptions to break. The current design preserves a useful reported string and parses machine data through explicit markers such as `FRAMESHIFT_DEVICES=` and

`FRAMESHIFT_METADATA=` . Machine protocols should use owned markers and JSON, not scrape incidental banners more than necessary.

Compressed .blend headers

The first file check assumed every valid scene began with the seven ASCII bytes `BLENDER` . Valid gzip- or Zstandard-compressed Blender containers therefore looked malicious. The fix accepts the uncompressed signature plus gzip `1f 8b` and Zstandard `28 b5 2f fd` , while still relying on bounded copying and sandboxed Blender inspection for semantic validation. Magic bytes are an admission check, not a complete trust decision, and file-format research must include supported container variants.

Context and working-directory errors

Native process code exposed two easy mistakes. A timeout context must be the one passed to `exec.CommandContext` ; deriving it and then launching with the parent context silently discards the deadline. Relative render assets also depend on a meaningful working directory. Invoking Blender from whichever directory started the server can make valid relative links fail. The durable fix is explicit context propagation, generated absolute managed paths, and a deliberate command directory where scene-relative resolution requires one. Process correctness includes environment and current directory, not only executable and arguments.

The old worker orphan

Restarting a worker under a changed identity can leave the old database row behind. Heartbeat aging correctly marks it stale and then offline, but the fleet still shows an orphan until an administrator removes it. More dangerously, process restarts must not let late completion from the old worker overwrite reassigned work. Lease ownership checks solve the correctness problem; health labels and safe remove controls solve the operational one. A future registration protocol should add instance IDs and retirement so identity changes are explicit rather than inferred from silence.

Migration timestamps

SQLite and PostgreSQL do not return time values in identical representations. A migration or scan path that assumes one timestamp encoding can work locally and fail against `TIMESTAMPTZ` , especially around text parsing, UTC normalization, or zero values. The fix was to use dialect-aware schema types, pass `time.Time` through the driver, and validate the lifecycle against real PostgreSQL rather than relying only on SQL string transformation. Database portability is behavioral, not textual. Placeholder rebinding and `DATETIME` replacement are necessary but insufficient evidence.

Together these bugs argue for boundary-focused tests. Git generators, native process output, binary signatures, context cancellation, worker liveness, and database drivers all sit where one subsystem's assumptions meet another's behavior. Those are exactly the places where a small executable check has the highest value.

Educational Lessons

The first lesson is that product UI should expose domain truth, not smooth it away.

RECOVERING , PAUSING , FINALIZING , stale workers, retries, and delivery approval are meaningful states. Replacing them with a generic spinner would make the interface simpler while making operations less trustworthy.

Second, local-first does not mean single-process. Even one machine contains a browser, API, database, worker, Blender, FFmpeg, filesystem, and optional services. Treating those boundaries seriously at local scale creates an architecture that can later cross hosts without changing who owns state.

Third, security and usability often reinforce each other. Immutable scene versions make review understandable. Disabling Blender auto-execution protects the host and gives inspection a reproducible baseline. Approval-gated publishing makes delivery auditable. Session revocation gives users a clear response to credential change.

Fourth, observability is a feature. Persisted logs, attempt history, progress phases, worker health, reports, and notifications are useful to operators because they are already useful to tests and recovery logic. The best live console is not theater layered over a child process; it is a readable projection of durable evidence.

Fifth, mocks and integration tests answer different questions. A mocked Playwright recovery can deterministically prove that a person sees and can act on worker loss. A Go scheduler test proves the lease and attempt transition. A PostgreSQL test proves driver and dialect behavior. A real Blender run proves the native boundary. Confidence comes from composing those claims without asking one slow test to prove everything.

Finally, documentation is part of validation. Explaining why a browser cannot authoritatively cancel a frame, why a compressed header is accepted, or why PostgreSQL tests are opt-in forces hidden assumptions into reviewable language. When the explanation is difficult, the boundary is often still confused.

Roadmap

The next product work should deepen the current model rather than replace it:

- Split the dashboard into feature-owned components and add focused React tests while preserving backend authority.

- Encode dashboard views, selected jobs, and selected scenes in routes or search parameters for deep links and browser history.
- Add focus management, keyboard-only journey tests, screen-reader review, and stricter axe thresholds beyond serious and critical findings.
- Add full-stack Playwright smoke tests against the Go API and SQLite, plus dedicated CI jobs that require PostgreSQL and MinIO tests to run rather than skip.
- Add concurrent scheduler tests for PostgreSQL claims, fairness, priority, `not_before`, finalizer lease expiry, and sustained SQLite contention.
- Improve fleet identity with worker instance IDs, explicit retirement, restart visibility, and confirmation around destructive administration.
- Expand scene review with sampled frames, image comparison, annotations, render-setting diffs, and richer missing-asset guidance.
- Turn Outputs into a searchable multi-job library with checksum verification, pagination, lifecycle confirmations, publication retry detail, and webhook test delivery.
- Replace broad polling where measured load justifies it, while retaining reconciliation so events are never the sole source of truth.
- Add visual regression coverage at desktop, tablet, narrow mobile, reduced motion, long content, offline API, and permission-denied notification states.

There are also architectural thresholds to watch. The global CSS and single dashboard module are manageable now but will become expensive as workflows multiply. The SQLite scheduler is appropriate for local scale but not a promise of high write concurrency. The current ETA is an operational estimate, not a deadline model. The thumbnail review is a production gate, not a complete creative-review suite. Naming these limits is healthier than hiding them behind polished UI.

Unified Conclusion

Across ten parts, FrameShift moved from a local-first idea to a product-shaped system. The early foundation established why render work must be managed, why Go, Next.js, SQLite, and native Blender fit their assigned responsibilities, and why the backend and database must remain authoritative. Secure ingestion then treated scene files and archives as hostile until bounded checks and constrained inspection produced immutable evidence. Durable scheduling turned frames into leased tasks with attempt history, control intent, recovery, progress, and finalization.

The later product concerns extend the same logic through identity, workspaces, reviews, notifications, artifacts, retention, backup, publishing, optional PostgreSQL and object storage, and finally the interface described here. None of those capabilities stands alone. Authentication determines who may act. Workspace roles constrain review and rendering. Approved immutable versions become durable jobs. Workers report evidence under leases. Recovery

preserves history. Finalization creates checksummed artifacts. Publishing performs a separately approved external side effect. Backups and retention keep the installation operable over time. The UI makes the chain visible and actionable.

Testing ties all ten parts together. Unit tests protect parsers and transitions. Store and HTTP tests protect authority and permissions. PostgreSQL, MinIO, Blender, and backup checks test real boundaries. Playwright tests protect the user journey, responsive behavior, recovery language, and accessibility. Incremental validation connects them into a development method that can locate failure instead of merely announcing it.

The central lesson is simple: infrastructure becomes a product when its guarantees are understandable at the moment a person needs them. FrameShift's dashboard does not conceal the machinery. It gives leases, attempts, review gates, worker health, artifacts, and recovery a humane operational shape while leaving final authority where it belongs. That continuity, from one local render to the complete tested interface, is what makes the ten-part system coherent.

FrameShift's ten parts describe one continuous system rather than a collection of features. Untrusted bytes become immutable scene versions; approved versions become durable frame tasks; leased workers turn those tasks into validated evidence; finalization turns evidence into checksummed deliverables; identity, policy, retention, publishing, and backups preserve their meaning. At every stage, useful specialist tools remain replaceable edges around one authoritative transactional center.

The broader lesson is that local-first software still deserves production-grade boundaries. A browser refresh, worker crash, laptop sleep, database change, interrupted upload, failed encode, or unavailable webhook should land in explicit state rather than folklore. Starting with one Mac made those boundaries easier to inspect. Preserving them is what allows the same design to grow into a render operations platform without losing the ability to explain what happened.