N Nightship

Build distribution your team — and your data — can trust.

A technical and strategic overview of Nightship: how pre-release builds move from developers to their teams, on infrastructure you own.

White paper · **Version 1.1** · 2026
Smartfull Solutions GmbH · “Nightship” is a working title

Abstract

Teams that build desktop software — game studios above all — ship pre-release builds to themselves dozens of times a day, yet the tools for doing so are ad-hoc: shared drives, chat attachments, bespoke scripts, or consumer platforms designed to reach players rather than colleagues. Nightship is a build-distribution system that treats internal delivery as a first-class problem. It moves builds through named *channels*, keeps every subscribed machine current through a quiet tray client, and lets testers *launch* the build with one button. Crucially, it separates the control plane (metadata, channels, permissions) from the data plane (the build bytes), so builds flow directly between machines and storage you own — never through a vendor's servers, never metered by the gigabyte. This paper describes the problem, Nightship's architecture, its two transfer engines — **Cargo** and **Tide** — and where it sits relative to the alternatives.

1. The problem: getting your own build to your own team

Distribution to *customers* is a solved, crowded space. Distribution to *your own team* — the nightly, the QA candidate, the build a producer needs to show a publisher this afternoon — is not.

A modern game or application build is large, changes constantly, and needs to reach a specific, trusted set of people quickly and correctly. The everyday questions are mundane and expensive: *Which build is current? Did everyone get it? Which version was "the one that broke"? How do I run it?* Teams answer these with a patchwork — network shares that offer no history or verification, chat uploads that lose track of versions, homegrown sync scripts that no one owns, or consumer storefronts whose entire model is reaching anonymous players, not a named internal team.

Three properties make internal build distribution genuinely hard, and are the requirements Nightship is built around:

- **Scale & churn.** Multi-gigabyte builds produced many times a day. Moving the whole thing every time wastes hours and bandwidth; moving *only what changed* is the difference between minutes and coffee breaks.
- **Correctness & provenance.** Everyone must be able to see and trust exactly which build is live on which channel, roll back instantly, and reproduce "it broke between build 44 and 47" without guesswork.
- **Sovereignty.** Unreleased builds are among a studio's most sensitive assets. For many teams they must never leave controlled infrastructure — and cost must not scale punitively with the bytes moved.

2. What Nightship is

Nightship is, in one line, a system that moves pre-release builds from developers to their teams — the internal counterpart to a storefront. Developers push builds manually or from CI and control which build sits in which **channel** (nightly, QA, RC, publisher). Team members **subscribe** to any number of channels; a tray **client** keeps every subscribed build current automatically, side by side, one install directory per subscription. And because the point of a build is to be run, Nightship is also the **launcher** testers open to start it.

The core nouns

- **Channel** — a named, movable pointer to an immutable build, exactly like a branch points to a commit. Promoting and rolling back are instant, atomic, and auditable.
- **Build** — an immutable, content-addressed snapshot with a signed manifest. Once finalized it never changes; a channel simply points at it.
- **Subscription** — a channel a machine follows, installed in its own directory so nightly, RC, and last-known-good can coexist untouched.
- **Launch target** — an admin-defined, per-channel command surfaced as a "Run" button in the client, so a tester never has to hunt for the executable.

> **Nightship already ships Nightship.** Every green build of the product itself is pushed through Nightship's own channel and lands on the founders' machines automatically. The product is its own permanently running demo — and its first customer.

3. Architecture: control plane and data plane, split by design

The single most important architectural decision in Nightship is that **build bytes never touch its servers.**

Nightship separates two concerns that most systems conflate. The **control plane** — channels, permissions, manifests, and short-lived access grants — runs on a Nightship server. The **data plane** — the actual build content — flows directly between a client and an object store that *you* own or that is provisioned for you at cost. The server hands out signed manifests and pre-signed URLs; the bytes go point-to-point.

Control plane (Nightship server)	Data plane (your storage)	Consequence
Channels, users, roles, groups, invites; manifests; pre-signed URL minting.	Content-addressed chunks and immutable manifests, in your S3-compatible bucket (Cloudflare R2, AWS, MinIO, Garage…).	No egress in Nightship's cost model; builds never transit a vendor. "We profit from the product, not your bytes."

The filesystem is the source of truth

Every entity — org, user, channel, build, token grant — is a small JSON file, written temp → fsync → atomic rename. Manifests are content-addressed JSON documents; chunks are content-addressed files; channels are tiny pointer files, like Git refs. A SQLite database is used only as a *disposable index* that can be rebuilt from the JSON tree at any time. The practical payoffs are large: a backup is a directory copy; migrating between self-hosted and cloud is copying that directory in either direction; and there is no database server to operate at any tier.

Trust the manifest, not the storage

Every manifest is signed with an Ed25519 key. A client verifies the signature before it trusts a single byte, which means the storage layer never has to be trusted — only *available*. Any static file server, mirror, or

CDN is a valid read path, because integrity is guaranteed by the signature rather than the transport. This is what makes “bring your own bucket” safe.

4. Two engines: Cargo and Tide

Every channel chooses how its builds travel. The client and the workflow are identical; only the wire efficiency differs.

DEFAULT

Cargo — whole-file transfer

Cargo moves builds file by file: simple, transparent, and rock-solid. Unchanged files are skipped; downloads stage into a scratch directory and swap into place atomically, with lock-aware updates that wait politely while a build is running and then finish themselves. Cargo is the dependable baseline — nothing exotic, nothing to tune.

DELTA

Tide — content-defined delta

Tide breaks each file into content-defined chunks (FastCDC boundaries, BLAKE3 chunk identities), compresses each chunk, and uploads only chunks the store doesn’t already hold. A new build’s manifest lists its chunks; a client fetches only the ones it is missing. Successive builds move in megabytes, not gigabytes — and dedup is global across every build and channel in the bucket.

Because chunk identities are computed over plaintext, compression is invisible to deduplication, and because boundaries are content-defined rather than fixed-size, an inserted or removed byte re-syncs quickly instead of shifting every subsequent block. Uploads are delta too: a push chunks the build locally, asks the store which chunks are new, and sends only those. The manifest is signed on the way up; a corrupted or partial install is repaired simply by re-fetching whatever chunks fail verification, from any starting point.

> **On the roadmap: container-aware chunking.** The highest-value fidelity step is cutting chunk boundaries at asset and compression-block boundaries inside game-engine containers (Unreal `.pak`/IoStore), so that a one-asset change touches one region rather than reshuffling a whole archive. Unknown formats fall back to generic content-defined chunking — a buggy parser degrades efficiency, never correctness.

Zero standing credentials on the client

For Tide channels the client holds *no* storage keys. When a machine syncs, the Nightship server mints short-lived pre-signed URLs for exactly the manifest and chunks that machine needs; the client reads directly from the bucket with nothing but those URLs, and verifies the signed manifest on arrival. A leaked client can leak nothing, because it was never trusted with a credential.

5. Security & sovereignty

Nightship treats unreleased builds as the sensitive assets they are, and its security model is deliberately boring in the right places.

- **Signed builds.** Ed25519-signed manifests mean storage need only be *available*, not trusted; every download is hash-verified before it is activated.
- **Scoped CI credentials.** A dedicated publisher role can do exactly three things — upload a build, finalize it, and promote it to an allow-listed channel. A leaked pipeline token can produce a junk build in one permitted channel — visible, revocable, recoverable — and can never touch users, other channels, or settings.
- **No credentials created behind your back.** A fresh server mints no tokens on its own; local access to the data directory is the root of trust, and administrative credentials are shown once and stored only as hashes.
- **Air-gap by design.** Nothing in the core loop phones home. Licensing is offline — a signed file and a baked-in public key — so Nightship runs on networks with no outbound access at all.
- **Your data is portable, always.** Your whole account exports as a plain folder a self-hosted server reads directly. There is no lock-in to engineer around; leaving is a copy.

6. Deployment & operations

Nightship ships as a single self-contained server binary that spans a spectrum of tiers from one configuration surface: a developer's workstation, a trusted LAN box, a private VPS behind HTTPS, a dedicated managed instance, or a fully managed cloud (arriving during the beta). The Linux server is published as an official Docker image, `ghcr.io/nightship-io/nightshipd` — public and ready to pull. Moving between hosted and on-premises, in either direction, is copying a directory.

Everything the UI can do exists as a documented CLI verb against an OpenAPI interface — JSON output, honest exit codes, CI-friendly scoped tokens — so automation is a first-class path, not an afterthought. The tray client (Windows today, macOS on the way) requires no administrator rights, patches into a staging directory and swaps atomically, verifies every hash, and offers a one-click "Restart to update" for the app it just launched.

7. Getting started: from nothing to nightly builds

Adopting Nightship is an afternoon, not a project. What follows is the path a studio actually walks, in order.

Try it first, on your own machine

Drydock runs a complete Nightship — server and storage — in Docker on a single workstation, from one button in the client under **Settings → Drydock**. It starts the containers, provisions a bucket, claims that server as your admin, creates a channel, pushes a first build and subscribes you to it. What you end up with is the whole producer-to-consumer loop on one tab: change a file in the build folder, push, and watch it arrive in the synced folder seconds later. It binds to localhost and uses fixed demo credentials, so it is for evaluating rather than production — but it is the fastest way to see the loop.

1. Start the server and mint the first admin

A fresh server has no accounts and answers every request with 401. There is no sign-up page and no default password, by design: local write access to the data directory is the root of trust. Whoever has it runs two offline subcommands, and the token is printed once, at creation.

```
nightshipd user create admin --role admin --data-dir /var/lib/nightship
nightshipd token create bootstrap --user admin --data-dir /var/lib/nightship
```

Add the server in the desktop app with that token; everything after this happens over the API.

2. Connect storage

A Tide channel keeps its builds in your bucket, and on Cloudflare R2 Nightship does the setup itself. Three things need your account and your card first — create the Cloudflare account, enable R2, and mint one API token, which the CLI takes from the environment rather than the command line.

```
export CLOUDFLARE_API_TOKEN=<the token you just copied>
nightship storage init          # reports what it would do, creates nothing
nightship storage init --yes    # creates the bucket and credentials
```

The preview names the account it found, suggests a bucket name and shows the endpoint your server will use. The run then creates one bucket — in the EU jurisdiction by default, and **data residency cannot be changed after the bucket exists** — and two scoped credentials, read-only for the server and read-and-write for CI, both locked to that one bucket. It saves the server's key and prints CI's credential once. Nightship stores none of it, and a failure part way through undoes whatever the run created. Any other S3-compatible store — AWS, MinIO, Garage — is connected by hand from **Admin → Storage**.

3. Create the channel

```
nightship channel create latest --server https://builds.example.com
```

Cargo is the default engine. Tide is chosen when the channel is created and is locked once builds exist, so the engine is a decision to make with the channel rather than after it.

4. Publish from CI

CI gets an identity of its own — not an admin token, and not a person's. A service account is a publisher nobody logs in as, restricted to the channels it publishes, and it does not consume a seat.

```
nightship user create ci --service
nightship user set-role ci publisher --channels latest
nightship token create ci-token --user ci
```

On GitHub the push is a single step. The action detects the runner's operating system, downloads the matching tide binary and runs the push; the bucket write key and the publisher token are the only secrets it needs, and the build bytes travel from the runner straight to your bucket without passing through the server.

```
- uses: nightship-tools/tide-push@v1
  with:
    dir: ./package
    channel: latest
    register-url: https://builds.example.com
  env:
    TIDE_S3_ENDPOINT: ${{ secrets.TIDE_S3_ENDPOINT }}
    TIDE_S3_BUCKET:   ${{ secrets.TIDE_S3_BUCKET }}
    TIDE_S3_KEY:      ${{ secrets.TIDE_S3_KEY }}
    TIDE_S3_SECRET:   ${{ secrets.TIDE_S3_SECRET }}
    TIDE_REGISTER_TOKEN: ${{ secrets.NIGHTSHIP_PUBLISHER_TOKEN }}
```

Any other CI does the same: fetch the tide binary for the runner and call push.

5. Bring the team in

Two routes, depending on whether the studio has an identity provider. Without one, use invites: a group carries channels and an invite carries groups, so a new member is following the right builds the moment they redeem it. Codes are single-use, time-boxed, and revocable before use.

```
nightship group create testers
nightship group assign-channel testers latest
nightship invite create teammate --groups testers --days 7
```

With one, use single sign-on. Register Nightship as an OpenID Connect application with Microsoft Entra ID, Okta, Keycloak or any OIDC provider, and people sign in with the account they already have. Role comes from a mapped application role and group membership from the provider's groups claim, both refreshed at every sign-in. Removing the person at the provider, or in Nightship, cuts access immediately. CI is unaffected: service accounts keep their tokens, because SSO is for people.

6. Turn on backups

Backups are opt-in — nothing is written anywhere until you set them up. The server chunks, compresses and encrypts the data directory before any of it leaves (XChaCha20-Poly1305), so the bucket never sees plaintext, and snapshots share unchanged chunks, so a daily backup costs only what changed. It can share the bucket Tide already uses, under a `backups/` prefix, with no second credential to manage.

```
nightship backup init --shared
```

The encryption key is generated by the server and shown exactly once; store it where you keep your other secrets, because without it a snapshot cannot be restored by anyone, ourselves included. The data directory holds everything the server knows, and for Cargo channels the builds themselves; Tide chunks live in your bucket, so its durability settings are the other half of the plan.

> **Two decisions you cannot take back.** A bucket's jurisdiction is fixed at creation, and a backup without its key is unrecoverable. Everything else on this list can be changed later.

8. Where Nightship fits

Nightship is not a storefront, not a version-control system, and not a shared drive. It is the missing layer between them.

Approach	What it is	Why it isn't this
Consumer storefronts (Steam & branches)	Distribution to anonymous players, with fixed-size chunking and store-shaped workflows.	Built to ship to customers, not your team; not self-hostable; the internal-distribution use is a workaround.
Managed build-delivery clouds	Hosted delivery for game teams, cloud-only, with metered egress.	Your builds live on a vendor; egress is billed by the gigabyte; no self-host or air-gap.
Version control (Perforce, Git-LFS)	Source-of-truth for source and large assets.	Manages the inputs to a build, not the distribution of the built artifact to testers, with channels and a launcher.
File shares & scripts	SMB drives, chat uploads, homegrown sync.	No history, no verification, no delta, no launch surface, no provenance — the status quo Nightship replaces.

Nightship's distinguishing stance is **sovereignty without a bandwidth tax**: self-hostable at every tier, storage you own, no metering of the bytes your team moves. Its commercial model follows the same logic — you pay only for the handful of privileged seats that operate the server and publish builds; every tester, playtester, and teammate who installs or runs a build is free and unlimited. Egress and storage are never a silent margin line.

9. Summary

Getting a build to your own team is a distinct problem from getting a product to the world, and it deserves a purpose-built tool. Nightship provides one: channels that behave like branches for builds, a quiet client that keeps every machine current and lets testers run the build with one button, and an architecture that keeps the bytes on infrastructure you control. Two engines — the dependable Cargo and the delta-efficient Tide — let each channel trade simplicity against wire efficiency without changing the workflow. Signed manifests, scoped credentials, offline licensing, and folder-level portability make it trustworthy with a studio's most sensitive assets. The result is a system that answers "where's the build?" once, and then gets out of the way.

© 2026 Smartfull Solutions GmbH. "Nightship" is a working title. This document describes a product under active development; roadmap items are identified as such. For evaluation or beta access, contact info@nightship.io or visit nightship.io.