

GAMEHUB / LEARN — FIELD NOTES

GameHub

A Futuristic Gaming Platform Built with AWS & DevOps

A build log across every phase of the project — what was built, why that choice beat the alternative, the bugs found, and the concepts worth understanding cold.

Generated from `README.md` and `docs/*.md` by `scripts/build-learn.js`.

Live: <https://gamehub.worldofvpc.online>

Contents

OVERVIEW GameHub

STEP 0 Step 0 – Reproducible Local Foundation  [2026-10-18]

STEP 1 Step 1 – Automated Tests + GitHub Actions CI  [2026-10-18]

STEP 2 Step 2 – Game-y UI: Live Background + Self-Hosted Arcade Fonts  [2026-10-18]

STEP 3 Step 3 – Real Playable Games  [2026-10-19]

STEP 4 Step 4 – Real Auth + Live on AWS  [2026-10-19]

STEP 5 Step 5 – Observability  [2026-10-19]

STEP 6 Step 6 – Local Kubernetes [kind] + Argo CD  [2026-10-19]

STEP 7 Step 7 – Security hardening & GDPR  [2026-10-19]

STEP 8 Step 8 – /learn Documentation Site + PDF Export  [2026-10-19]

DOC Architecture decisions [ADRs]

DOC Load test – what a single t3.micro actually does  [2026-10-19]

DOC Runbook – GameHub [prod]

DOC Service level objectives – GameHub

A Futuristic Gaming Platform Built with AWS & DevOps

CI License: MIT Tests Coverage

GameHub is a portfolio project that combines a futuristic gaming interface with a containerized Node.js backend and a MySQL database – all runnable locally with one command. It demonstrates how frontend applications, backend APIs, container networking, reverse proxies, databases, tests and CI pipelines work together.

The project is being evolved into a production-style application: automated CI/CD, infrastructure as code on AWS, monitoring, and Kubernetes (local kind cluster + Argo CD).

Status – Phase 8 complete (2026-10-19): tested, CI'd, game-y, playable and **live on AWS**, observable, runnable on Kubernetes, hardened – and documented. 84 automated tests, a Cognito sign-in gate with **working account deletion** (GDPR art. 17, in the app rather than in a policy document), a bilingual privacy notice, an SBOM + image scanning in CI, deploys that only run on a green CI for the exact commit, a measured `t3.micro` load test, and the whole write-up published at `/learn/` as a site and a PDF. All described as CloudFormation and deployed by keyless GitHub OIDC, on a custom domain with an ACM certificate.

Live demo: <https://gamehub.worldofvpc.online> – create an account (email confirmation), sign in, and play. The landing page is public; the catalogue and the games require a signed-in player.

Documentation: the step-by-step write-up, the ADRs and the runbook are published as a small static site at <https://gamehub.worldofvpc.online/learn/>, with the whole guide as a printable PDF, and every claim in it traceable to a commit and a CI run.

Project Status

Current phase: Phase 8 – `/learn` documentation site, PDF & load test  (see `docs/step-8-learn-site.md` and `docs/load-test.md`)

Status: Full stack runs locally end-to-end: `make bootstrap` → seeded database, healthy containers, passing smoke tests. No cloud resources required.

Completed (Phase 0)

- Futuristic gaming homepage
- Dedicated Games page
- Dynamic game catalogue loaded through a backend API (12 seeded records)
- Game category filters
- Frontend built with HTML, CSS, and JavaScript
- Node.js and Express backend

- **Local MySQL 8.4 container** with schema + seed [`db/init.sql`] – reproducible on any machine, no cloud account needed
- **Connection pool** (not a connection-per-request), **liveness/readiness split** [`/api/health` vs `/api/ready`], structured JSON logs, CORS locked to explicit origins, generic client-facing errors
- Dockerized frontend and backend on the **Alpine stable channel** (Dependabot-tracked, security-patched at build time), `npm ci` from lockfile, **0 npm vulnerabilities**, backend running as **non-root**
- Docker Compose orchestration with **healthchecks + ordered startup**
- Nginx reverse proxy (gzip, security headers) – backend port **not published** to the host
- `make bootstrap` one-command workflow + **smoke test** [`scripts/smoke.sh`]
- Git and GitHub repository setup

✅ Completed (Phase 1 – tests & CI)

- **14 automated tests** using Node's built-in `node:test` (zero extra test deps): unit suite (runs anywhere, no DB) + integration suite against real MySQL (skips loudly if no database is reachable)
- Behavioural coverage of the paths that matter: liveness survives a database outage while readiness returns 503, CORS refuses unknown origins, driver errors never leak to clients, malformed JSON → 400
- **Coverage gate**: ≥80% lines / branches / functions or CI fails
- **ESLint** (flat config) as a build gate
- **GitHub Actions CI – 3 jobs on every push/PR**:
 1. lint + tests + coverage gate against a MySQL **service container**
 2. **full-stack smoke test** – builds the real Docker images, runs the real compose stack on the runner
 3. **security scans** – gitleaks (full history), Trivy (CRITICAL/HIGH), npm audit
- Local equivalents: `make test` · `make lint` · `make coverage`

✅ Completed (Phase 2 – game-y UI)

- **Live animated background**: canvas starfield (3 parallax layers, pixel stars), rising neon pixel particles, rare shooting stars – plus a scrolling grid and a drifting neon orb. Zero JS dependencies, ~118 draw calls/frame
- **GDPR-clean fonts**: Press Start 2P (arcade accents) + Chakra Petch (headings/body) **self-hosted** as woff2 with OFL licenses in-repo – the Google Fonts CDN import is gone, so **zero third-party requests** at load
- **Accessibility**: `prefers-reduced-motion` visitors get a fully static page (canvas is never even created)
- Arcade styling on all interactive elements (PS2P buttons with press feel)

✅ Completed (Phase 3 – real playable games)

- **All 12 catalogue games are actually playable** – `play.html?id=<n>` fetches the title and runs one of **6 dependency-free canvas engines**: snake, breakout, survival-racing (dodge), memory, reflex targeting, pong
- Runner state machine: READY → PLAYING → GAME OVER with **Play again**, HUD (score / best / status), per-title **high scores in localStorage** (guarded), controls help, Escape → catalogue
- **GET /api/games/:id** endpoint: honest 404s for unknown *and* non-numeric ids; malformed/oversized JSON now 400/413 instead of 500
- **Frontend tests (21)**: the six engines execute in Node against a stubbed canvas – frame-driven, asserting real terminal paths; plus a **coverage contract test** that fails CI if any seeded title lacks an engine
- An engine test found and fixed a real stall bug (dead-centre breakout hit → vertical ball → unwinnable game)

✓ Completed (Phase 4 – real auth + live on AWS)

- **Cognito sign-in gate:** sign-up with email confirmation, sign-in, sign-out. The static frontend talks to Cognito's REST API directly (no Amplify, no SDK) and sends the ID token as a `Bearer` header on every `/api/*` call
- **Server-side token verification** (`backend/auth.js`): RS256 signature checked against the pool's JWKS with `node:crypto`, plus `iss`, `aud / client_id`, `exp` and `token_use` – no `jsonwebtoken` dependency
- **Honest fail-open:** the gate is enforced only when the pool is configured; local development runs open and the backend logs an `auth_disabled` warning at boot – never a silent bypass
- **Static site on S3 + CloudFront:** private bucket, CloudFront Origin Access Control, `/api/*` behaviour with caching disabled, HTML short-cache vs assets long-cache
- **API host on EC2** running the *same* `docker-compose.yml` as local development (nginx + Express + MySQL), with a 2 GB swap file for headroom
- **No SSH:** the security group opens port 8080 to the AWS-managed CloudFront origin-facing prefix list only; administration is via SSM Session Manager, IMDSv2 is required and the root volume is encrypted
- **Four CloudFormation stacks** (`infra/cognito.yaml`, `site.yaml`, `backend.yaml`, `github-oidc.yaml`) – one command per stack, outputs wired into the next
- **Keyless deploys:** GitHub Actions assumes a scoped role through OIDC (no AWS access keys anywhere); the workflow publishes the site, invalidates CloudFront and rolls out the API host over SSM
- **Landing page stays public** while the catalogue and the runner require a signed-in player – found by actually signing out live and following the redirect, not by reading the code
- **Versioned asset URLs** (`script.js?v=<deploy sha>`): assets are cached for a week, so a CloudFront invalidation alone still leaves returning browsers on the previous script. HTML (one-minute cache) points at the new version – found by watching a stale `script.js` run in the browser after a deploy

✓ Completed (Phase 5 – observability)

- **Structured request logs:** nginx emits JSON access logs (`status`, `uri`, `request_time`, `upstream_time`) – CloudWatch parses them without regex guesswork
- **CloudWatch agent on the host:** ships those logs plus container stdout to `/gamehub/prod/nginx-access` and `/gamehub/prod/app` (7-day retention set by the template), and host `mem / disk / swap` metrics into the `CWAgent` namespace – installed by user-data, no manual setup
- **Metrics derived from logs** (`AWS::Logs::MetricFilter`): request count, 5xx count, and API latency as a *numeric* extraction (`$.request_time`) for p50/p99 – nginx already knew the numbers, so the app is not instrumented with an extra library to repeat them
- **External availability check:** EventBridge runs a small Lambda every 5 minutes against the public CloudFront URL and `/api/health`; its log line becomes the `SyntheticSuccess` metric – a synthetic check without the CloudWatch Synthetics price tag
- **Eight alarms** into one SNS topic: synthetic availability (`TreatMissingData: breaching` – silence is a failure), 5xx count, API p99 latency, instance status, CPU, CPU credits, memory and disk. Host-metric alarms treat missing data as `notBreaching`, because a stopped agent is not an outage
- **One dashboard** (`gamehub-prod`): availability, requests vs errors, latency percentiles, host pressure, and two live log tables (recent 5xx, backend tail)
- **SLOs with error budgets** (`docs/slo.md`): 99.5 % availability and $p99 \leq 1.5$ s over a rolling 30 days, with the measurement method for each – and an explicit list of what is **not** claimed, because a single-AZ t3.micro cannot honestly promise it

- **Runbook** [`docs/runbook.md`]: one section per alarm with the exact SSM/Logs-Insights commands to confirm the cause, plus the revert-to-roll-back path
- **Verified live, not assumed**: the agent's metric dimensions were read back from `list-metrics` and then written into the template – an alarm with the wrong dimension set sits in `INSUFFICIENT_DATA` forever

✓ Completed (Phase 6 – local Kubernetes + Argo CD GitOps)

- **Real cluster, no bill**: a single-node `kind` cluster [`k8s/kind-config.yaml`] with the same images the Compose stack builds, side-loaded into the node [`make kind-up`]
- **Deployments and a StatefulSet**: frontend and backend run 2 replicas each with `maxUnavailable: 0` rollouts; MySQL is a StatefulSet with a volume claim template – a Deployment would give the database a random name and two pods fighting over one data directory
- **Probes with consequences**: liveness on `/api/health` [no database], readiness on `/api/ready` [database] – a DB outage stops new traffic instead of restarting every pod
- **One source of truth for the seed**: `db/init.sql` is rendered into a ConfigMap by `scripts/build-k8s-dbinit.sh`, and CI fails if the copy drifts from the source
- **Least privilege, measured rather than assumed**: the backend runs as the unprivileged `node` user with `drop: ALL`; nginx keeps exactly the five capabilities its root master process needs – the first attempt at `drop: ALL` produced `CrashLoopBackOff` with `chown(...)` failed (1: Operation not permitted), which is why the list is now written down with a reason
- **GitOps with Argo CD** [`k8s/argocd/application.yaml`]: `automated` sync with `prune` and `selfHeal`, so the merge to `main` is the deploy and a manual `kubectl` edit is reverted by the controller
- **Verified like the rest of the project**: the same `scripts/smoke.sh` runs against the cluster's NodePort (8081, so Compose on 8080 keeps working) and passes – 12 games served from the StatefulSet through the nginx proxy
- **CI covers it**: a `manifests` job validates every manifest against the Kubernetes schema (kubecconform) and checks the generated ConfigMap for drift

✓ Completed (Phase 7 – security hardening & GDPR)

- **Erasure that is implemented, not promised**: `DELETE /api/me` deletes the player's rows **and** their Cognito identity in one action from the `account` page. A partial result is reported as a partial result [`502`], never as success
- **The account page is the GDPR surface**: it is the access/portability answer (identity + every stored row, copyable as JSON) and it carries the deletion control – one page, three of the four rights, no support email
- **A bilingual privacy notice** [`frontend/privacy.html`]: a data inventory with purpose, legal basis and retention per item – including the 7-day access logs, which are personal data too
- **One AWS call, signed by hand**: account deletion needs `cognito-idp:AdminDeleteUser`, so the backend fetches IMDSv2 credentials and signs the request itself (SigV4, `node:crypto`) – no AWS SDK, no long-lived keys, and the signature is asserted field by field in the test suite
- **Least privilege on the identity**: the instance role's inline policy allows that *one* action on *one* user-pool ARN, not `CognitoPowerUser` on `*`
- **Data minimisation in the schema**: one `personal_bests` row per (player, game) with `GREATEST()` on conflict – the best score, no play history – keyed by the Cognito `sub`, so the database holds no email address
- **Migrations with a ledger** [`scripts/migrate.sh`, `db/migrations/`]: `db/init.sql` only runs on an empty volume, so schema changes ship as checksummed, idempotent migrations; a rewritten migration is refused

- **Supply chain in CI:** SPDX SBOM (syft) kept as an artifact, Trivy against the **built image** as well as the filesystem, `npm audit`, gitleaks over the full history, and Dependabot for npm, Actions and base images
- **Deploys are CI-gated:** the deploy workflow runs on a *successful CI run for the exact commit* (never a later, untested tip of `main`) and finishes with a post-deploy smoke test against the public URL: the app shell, the documentation site and its PDF, a `401` from every protected route without a token, and – signing in as a dedicated smoke identity – the 12 catalogue entries, the game details, and an identified player from `/api/me`

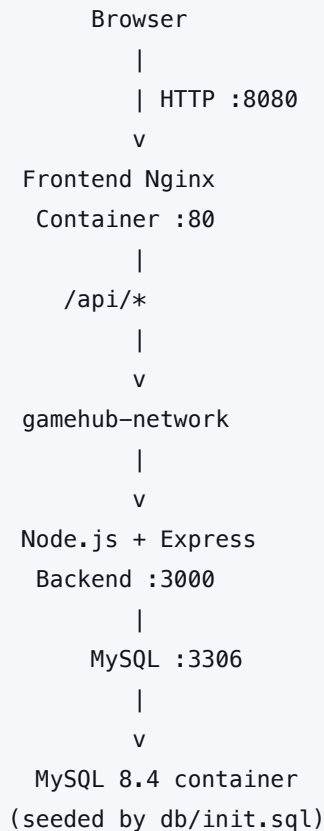
✓ Completed (Phase 8 – /learn documentation site, PDF, load test)

- **A documentation site generated from the repository:** `scripts/build-learn.js` (dependency-free, Node built-ins only) renders `README.md` and every `docs/*.md` into `learn/` – a landing page, one page per document, a print guide, and a hand-written Markdown renderer with GitHub-style table support
- **Published with the app:** `build-frontend.sh` copies `learn/` into the deployable artifact, so the *same* CloudFront distribution serves the guide at `/learn/` – no second bucket, no second CDN, and no chance of the docs being deleted by the root sync. `/learn/` and `/learn` are resolved by a twelve-line CloudFront Function rather than by a folder object in the bucket (ADR-017) – before it existed, `/learn/` answered 200 with the *game hub's* home page
- **A downloadable PDF:** `scripts/build-pdf.sh` prints the print-optimised `learn/guide.html` through the browser already on the machine (A4 @page CSS), producing `learn/gamehub-guide.pdf` for offline reading
- **A CI drift check:** the generated site is committed, so the `manifests` job regenerates it and fails on any diff, and asserts that every document has a page – a generated artifact that is never checked is an artifact that goes stale
- **A load test with a real report:** `scripts/load-test.js` (dependency-free, committed) drives a weighted mix of static, API and probe traffic; `docs/load-test.md` records the measured numbers for a single `t3.micro` – including a deliberately pushed run, what the alarms did, and an explicit list of what the test does **not** prove
- **Erasure verified by a script, not by hand:** `scripts/verify-gdpr.sh` creates a real Cognito user, stores a score, deletes the account through the public API, and asserts that both the row and the identity are gone – cleaning up after itself even when it fails
- **Architecture decisions in one file:** `docs/decisions.md` – 18 decisions in context / decision / consequences / rejected form, each one stating what it costs

⚠ Next (deliberately not started)

- A custom domain in front of CloudFront (certificate + DNS record; the distribution already exists)
 - Moving the database to RDS (multi-AZ, automated backups, managed patching) – the cost/ops trade-off
`docs/decisions.md` ADR-002 records
-

Architecture

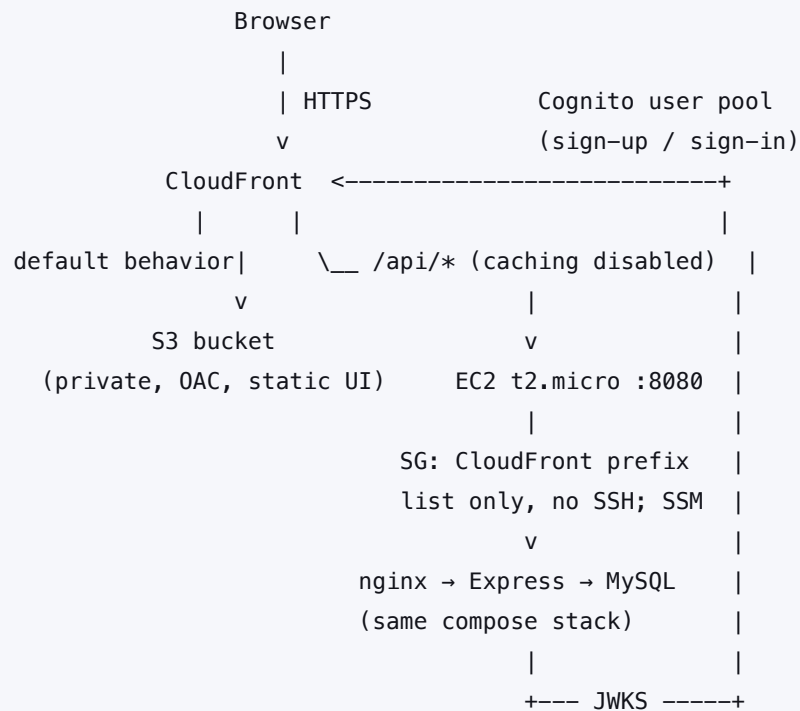


Request flow

1. A user opens GameHub in the browser.
2. Nginx serves the static frontend files.
3. JavaScript sends API requests to the frontend's `/api/` path.
4. Nginx forwards those requests to the backend container.
5. Express processes the request and queries the MySQL container over the private Docker network.
6. The backend returns JSON, which the frontend uses to render game cards.

Important: The browser accesses the frontend through port `8080`. The Nginx container communicates with the backend using Docker networking, and the backend port is **not published to the host** — the database runs as a local MySQL container [managed by hand, not by IaC — see the note in the Database section].

Live deployment (Phase 4)



The browser signs in against Cognito, receives an **ID token**, and sends it as `Authorization: Bearer <token>` on every API call. The backend verifies the token's RS256 signature against the pool's public JWKS – the frontend guard is UX, the API is the security boundary.

Technology Stack

Technology	Purpose
HTML5	Frontend structure
CSS3	Styling and responsive visual design
JavaScript	UI interactions, API requests, and category filters
Node.js	Backend JavaScript runtime
Express.js	REST API
MySQL 8.4	Relational database, local container (managed RDS via IaC in Phase 4)
Docker	Application containerization
Docker Compose	Multi-container orchestration
Nginx	Static web server and reverse proxy
Git	Version control
GitHub	Source code hosting
Colima	Local Docker runtime on macOS
Make	One-command developer workflow (<code>make bootstrap</code>)
Amazon Cognito	Player sign-up / sign-in, issues the ID token the API verifies
Amazon S3 + CloudFront	Static site hosting (private bucket, OAC) and global edge delivery
Amazon EC2	Hosts the compose stack (nginx + API + MySQL)
AWS CloudFormation	All infrastructure as code (<code>infra/*.yaml</code>)
AWS Systems Manager	Keyless administration and deploys run over SSM, without SSH
GitHub Actions OIDC	Keyless deploys: short-lived AWS credentials, no stored secrets

Project Structure

GameHub/

```
├─ frontend/
│   │   └─ index.html          ← landing page (hero, tournaments, leaderboard)
│   │   └─ games.html         ← catalogue (rendered from the API)
│   │   └─ play.html          ← game runner host (READY → PLAYING → GAME OVER)
│   │   └─ signin.html        ← Cognito sign-in
│   │   └─ signup.html        ← Cognito sign-up + email confirmation step
│   │   └─ profile.html       ← account page: stored data, JSON export, deletion
│   │   └─ privacy.html       ← bilingual (EN/DE) privacy notice
│   │   └─ script.js          ← homepage + catalogue logic
│   │   └─ play.js            ← runner: ENGINE_MAP, HUD, high scores
│   │   └─ engines.js         ← 6 dependency-free canvas game engines
│   │   └─ auth.js            ← Cognito REST client, session, page guard
│   │   └─ account.js         ← account page: fetch scores, export, delete
│   │   └─ config.js          ← runtime config (local dev: auth off)
│   │   └─ bg.js              ← animated arcade background
│   │   └─ style.css
│   │   └─ fonts/             ← self-hosted game fonts (OFL)
│   │   └─ nginx.conf
│   │   └─ test/              ← engine, mapping contract and auth tests
├─ backend/
│   │   └─ Dockerfile
│   │   └─ .dockerignore
│   │   └─ .env.example       ← copy to .env only for non-Docker runs
│   │   └─ db.js              ← connection pool
│   │   └─ auth.js            ← Cognito JWKS verification (no SDK)
│   │   └─ cognito.js         ← IMDSv2 credentials + SigV4 AdminDeleteUser
│   │   └─ app.js             ← Express app (exported for tests)
│   │   └─ server.js          ← HTTP endpoint
│   │   └─ test/              ← unit, integration, auth, GDPR and signer suites
│   │   └─ package.json
│   │   └─ package-lock.json
├─ db/
│   │   └─ init.sql           ← schema + seed (runs on first boot)
│   │   └─ migrations/       ← checksummed migrations applied to live schemas
├─ infra/
│   │   └─ cognito.yaml       ← user pool + app client
│   │   └─ site.yaml          ← S3 + CloudFront (+ /api/* origin)
│   │   └─ backend.yaml       ← EC2 host, security group, IAM, elastic IP
│   │   └─ observability.yaml ← log groups, metric filters, canary, alarms
│   │   └─ github-oidc.yaml   ← OIDC provider + scoped deploy role
├─ k8s/
│   │   └─ kind-config.yaml    ← single-node kind cluster (node port → host 8081)
│   │   └─ namespace.yaml
│   │   └─ config.yaml        ← ConfigMap (app config) + local-only Secret
│   │   └─ db.yaml            ← MySQL StatefulSet + headless Service + PVC
│   │   └─ db-init-configmap.yaml ← GENERATED from db/init.sql (CI checks drift)
```

```

|   ├── backend.yaml      ← Express Deployment + `backend` Service
|   ├── frontend.yaml    ← nginx Deployment + NodePort Service
|   └── argocd/
|       └── application.yaml ← Argo CD Application (prune + selfHeal)
└── scripts/
    ├── smoke.sh          ← end-to-end checks
    ├── build-frontend.sh ← builds the deployable static artifact
    ├── deploy-frontend.sh ← S3 sync + CloudFront invalidation
    ├── kind-up.sh        ← cluster + images + manifests + smoke test
    ├── argocd-up.sh      ← install Argo CD, sync the app, wait for Healthy
    ├── build-k8s-dbinit.sh ← renders the DB seed ConfigMap (--check in CI)
    ├── migrate.sh        ← applies db/migrations (ledger + checksums)
    ├── build-learn.js     ← builds the /learn site from docs/
    └── build-pdf.sh       ← renders the guide to PDF
└── learn/                ← generated documentation site + PDF
└── .github/
    ├── dependabot.yaml  ← weekly npm / Actions / base-image updates
    └── workflows/
        ├── ci.yaml      ← lint, tests, coverage, compose smoke, manifests, scans
        └── deploy.yaml  ← CI-gated OIDC deploy + post-deploy verification
└── docs/                ← learning notes, SLOs, runbook, ADRs
└── Dockerfile           ← frontend image (nginx)
└── docker-compose.yml   ← frontend + backend + db
└── Makefile             ← make bootstrap / up / smoke / test / db-reset
└── .dockerignore
└── .gitignore
└── README.md

```

Security note: no credentials are needed for local development – the defaults in `docker-compose.yml` are for the private Docker network only. If you run the server outside Docker, copy `backend/.env.example` to `backend/.env`. Real `.env` files are excluded from Git **and** from image builds [`backend/.dockerignore`].

Run GameHub Locally

Prerequisites

- macOS or Linux
- Git
- A Docker runtime + Docker Compose (on this Mac: [Colima](#) – start it with `colima start`)

No configuration needed. The stack ships with working local defaults: a MySQL container is seeded automatically from `db/init.sql` on first start.

1. Clone and bootstrap

```
git clone https://github.com/cijostany6-bit/GameHub.git
cd GameHub
make bootstrap      # build → start → wait for health → run smoke tests
```

2. Open GameHub

http://localhost:8080

Useful commands

```
make ps              # container status (healthy?)
make logs            # tail all logs
make smoke           # re-run the end-to-end checks
make test            # backend unit + integration tests (needs MySQL)
make test-frontend   # game-engine, mapping-contract and auth tests (Node only)
make down            # stop, keep database data
make db-reset        # wipe + re-seed the database
make clean           # full teardown incl. images and DB volume
make kind-up         # same app on a local Kubernetes cluster (kind) → :8081
make argocd-up       # install Argo CD and sync the manifests from Git
make k8s-dbinit      # re-render the Kubernetes DB seed ConfigMap
make learn           # rebuild the /learn docs site and the PDF
make migrate         # apply db/migrations to the running database
make load-test SITE_URL=https://... # dependency-free load test (see docs/load-test.md)
make verify-gdpr     # verify account deletion end to end (needs a deployment)
make help            # list all targets
```

Prefer plain Compose? `docker-compose up -d --build` then `./scripts/smoke.sh` does the same.

3. Or run it on Kubernetes

```
make kind-up        # kind cluster + images + manifests + the same smoke test
make argocd-up      # Argo CD takes over from Git: Synced + Healthy
```

The cluster publishes the app on **http://localhost:8081** so it can run next to the Compose stack on 8080. `make kind-down` deletes the cluster; the Docker images stay. Everything about it – including what it does *not* promise – is in `docs/step-6-kubernetes-gitops.md`.

Deploy to AWS

Everything is CloudFormation in `infra/` – six small stacks, each one a single command. Stack outputs feed the next stack, so there is no spreadsheet of ids to copy by hand.

```

export AWS_REGION=eu-central-1
export ZONE_ID=<your route 53 hosted zone id> # aws route53 list-hosted-zones

# 1. Cognito user pool + app client
aws cloudformation deploy --region "$AWS_REGION" \
  --stack-name gamehub-cognito --template-file infra/cognito.yaml \
  --parameter-overrides Stage=prod

# 2. Certificate for the custom domain. CloudFront only accepts certificates
#   from us-east-1, hence the separate stack and the separate region. DNS
#   validation happens inside this stack – there is nothing to click.
aws cloudformation deploy --region us-east-1 \
  --stack-name gamehub-dns --template-file infra/dns.yaml \
  --parameter-overrides DomainName=gamehub.worldofvpc.online HostedZoneId="$ZONE_ID"

# 3. S3 + CloudFront (deployed first without an API origin)
aws cloudformation deploy --region "$AWS_REGION" \
  --stack-name gamehub-site --template-file infra/site.yaml \
  --parameter-overrides \
    Stage=prod \
    CustomDomain=gamehub.worldofvpc.online \
    CertificateArn=<gamehub-dns CertificateArn output> \
    HostedZoneId="$ZONE_ID"

# 4. EC2 host running the compose stack (needs the CloudFront domain)
aws cloudformation deploy --region "$AWS_REGION" \
  --stack-name gamehub-backend --template-file infra/backend.yaml \
  --parameter-overrides \
    Stage=prod \
    CognitoUserPoolId=<pool id> \
    CognitoClientId=<client id> \
    SiteOrigin=<https://cloudfront domain> \
    DbPassword=<16+ chars> RootDbPassword=<16+ chars>

# 5. Re-deploy the site stack with the API origin attached, then publish.
#   Repeat every parameter: CloudFormation silently uses the defaults for
#   anything you leave out, and the defaults here would strip the custom
#   domain back off.
aws cloudformation deploy --region "$AWS_REGION" \
  --stack-name gamehub-site --template-file infra/site.yaml \
  --parameter-overrides \
    Stage=prod \
    BackendOriginDomain=<ec2 public dns> \
    CustomDomain=gamehub.worldofvpc.online \
    CertificateArn=<gamehub-dns CertificateArn output> \
    HostedZoneId="$ZONE_ID"

AWS_REGION=$AWS_REGION ./scripts/deploy-frontend.sh

```

```
# 6. Observability: log groups + retention, metrics from logs, the synthetic
# check, eight alarms and the dashboard. AlarmEmail is optional – without it
# the SNS topic is still created and can be subscribed later (see the
# AlarmTopicArn output), it just has no subscriber yet.
aws cloudformation deploy --region "$AWS_REGION" \
  --stack-name gamehub-observability --template-file infra/observability.yaml \
  --capabilities CAPABILITY_NAMED_IAM \
  --parameter-overrides \
    Stage=prod \
    InstanceId=<ec2 instance id> \
    SiteUrl=https://gamehub.worldofvpc.online \
    AlarmEmail=<you@example.com>
```

Deploy the observability stack **before** the host's CloudWatch agent writes its first log line: the template owns the log groups so their retention is set, and an agent-created group would make that resource fail with `AlreadyExists`.

For continuous deployment, deploy the fourth stack (`infra/github-oidc.yaml`) once, then set these GitHub repository variables and push to `main` – the `deploy workflow` does the rest through OIDC, with no AWS keys stored anywhere:

```
AWS_REGION , AWS_DEPLOY_ROLE_ARN , SITE_BUCKET , DISTRIBUTION_ID , INSTANCE_ID ,
COGNITO_USER_POOL_ID , COGNITO_CLIENT_ID .
```

The workflow does not run on the push itself. It runs when `CI succeeds on main` (`workflow_run + conclusion == 'success'`) and checks out the commit CI tested, so production never receives a commit that has not been judged. The rollout then pulls the revision on the host, rebuilds the containers, waits for MySQL and applies pending migrations (`./scripts/migrate.sh`) before a final job smoke-tests the public CloudFront URL – homepage, all 12 catalogue entries, game details and a `401` from an unauthenticated `/api/me`.

Cost note (honest): the site (S3 + CloudFront), Cognito and the four CloudFormation stacks are inside the free tier at this traffic level. The EC2 instance is the only line item: this account is on a free plan, so AWS *rejects* instance types that are not free-tier eligible (`t2.micro` failed with "not eligible for Free Tier") – the stack therefore defaults to `t3.micro`. On a non-free-tier account that is roughly \$8/month. Tear everything down with `aws cloudformation delete-stack` in reverse order.

Backend API

The following endpoints are implemented in the current application.

Method	Endpoint	Purpose
GET	<code>/api/health</code>	Liveness – process is up (never touches the database)
GET	<code>/api/ready</code>	Readiness – database reachable, safe to route traffic (503 otherwise)
GET	<code>/api/games</code>	Retrieve games from the database
GET	<code>/api/games/:id</code>	Single game (powers the runner); 404 for unknown <i>and</i> non-numeric ids
GET	<code>/api/me</code>	Echoes the verified Cognito identity – proof the token was checked
POST	<code>/api/scores</code>	Stores the player's best score for a game (upsert, <code>GREATEST</code>) – <code>201</code> with the stored best
GET	<code>/api/scores/me</code>	Every stored row for the caller: the access/portability answer
DELETE	<code>/api/me</code>	Erasure – deletes the caller's scores <i>and</i> their Cognito identity; <code>204</code> , or <code>502</code> if only half of it worked

The API is reachable **only through the nginx proxy on `:8080`** – port `3000` is deliberately not published to the host (use `docker-compose exec backend ...` if you need direct access from inside the network).

Auth behaviour: middleware on every `/api/*` route except the two probes. If the Cognito pool is configured (that is: in the deployment), requests without a valid ID token get `401 {"error":"Unauthorized"}`. If it is not configured (local development), the read-only routes keep working, the identity echo answers `401`, and the signed-in features answer `501` rather than pretending – the backend also logs `auth_disabled` at boot, so the bypass is never silent.

```
curl http://localhost:8080/api/health
curl http://localhost:8080/api/ready
curl http://localhost:8080/api/games
curl http://localhost:8080/api/games/4

# with the pool configured, protected routes need the ID token:
curl -H "Authorization: Bearer $ID_TOKEN" http://localhost:8080/api/me
curl -H "Authorization: Bearer $ID_TOKEN" http://localhost:8080/api/scores/me
curl -H "Authorization: Bearer $ID_TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"game":"Cyber Strike","score":42}' \
  http://localhost:8080/api/scores

# erasure (irreversible – it deletes the Cognito user as well):
curl -X DELETE -H "Authorization: Bearer $ID_TOKEN" http://localhost:8080/api/me
```

Or run the full check suite: `make smoke`.

Database

GameHub uses **MySQL 8.4 in a container**, seeded automatically from `db/init.sql` on the first start of an empty volume. This makes a fresh clone reproducible with no cloud account and no secrets – and the deployed stack runs the *same* container on the EC2 host, so there is exactly one database setup to reason about. Moving

to RDS (multi-AZ, automated backups, managed patching) is a deliberate cost/ops trade-off left for a later phase rather than a claim made here.

Reset to a clean seeded state any time with `make db-reset`.

The `games` table contains these fields:

Column	Purpose
<code>id</code>	Game identifier
<code>name</code>	Game title
<code>category</code>	Category used by the frontend filters
<code>description</code>	Short game description

Sample categories include Action, Racing, and Strategy.

Player data (`personal_bests`)

Column	Purpose
<code>user_sub</code>	The Cognito <code>sub</code> – never the email address
<code>game</code>	Game name (validated against <code>games.name</code> , so no free-text rows)
<code>score</code>	The best score only (<code>GREATEST()</code> on conflict)
<code>updated_at</code>	When that best was last improved

The primary key is `(user_sub, game)`: one row per player per game, which is the whole data-minimisation argument. Players are identified to the database by a pseudonymous Cognito `sub`; no email, no name, no IP is stored next to a score.

Migrations

`db/init.sql` runs **once**, on an empty data volume – which is correct for a fresh clone and wrong for a live database. Schema changes therefore ship as migrations:

```
./scripts/migrate.sh          # apply everything pending
./scripts/migrate.sh --status  # what is applied, what is pending
```

Each file in `db/migrations/` is recorded in a `schema_migrations` ledger with a **checksum**. A migration that was applied and then edited is refused, because the database that ran it is no longer described by the file. The runner reaches MySQL through `docker compose exec` on a host and through the `mysql` client in CI (`MIGRATE_VIA`), so the same script runs in both places. It is also part of the deploy: the rollout brings the schema up to date after the containers start.

The frontend retrieves game records dynamically through the Express API rather than relying entirely on hardcoded game cards.

Docker Design

Frontend container

- Uses Nginx Alpine.
- Serves HTML, CSS, and JavaScript.
- Loads the custom Nginx configuration.
- Forwards `/api/` requests to the backend service.

Backend container

- Uses Node.js Alpine.
- Installs application dependencies.
- Runs the Express server.
- Reads database configuration from environment variables.
- Connects to the MySQL container over the private Docker network (no host port published).

Docker Compose

Compose manages both services and connects them through the `gamehub-network` network.

Useful commands:

```
docker-compose ps
docker-compose restart
docker-compose up -d --build frontend
docker-compose up -d --build backend
docker-compose logs -f backend
docker-compose down
```

Rebuild a service after changing its code:

```
docker-compose up -d --build frontend
```

Or:

```
docker-compose up -d --build backend
```

Security Practices

- **No secrets required for local development** – compose defaults live in the private Docker network; nothing sensitive exists in the repo.
- `.env` is excluded from Git **and** from image builds (`backend/.dockerignore`); `backend/.env.example` documents every variable.
- The backend runs as a **non-root** user and reads all configuration from environment variables (12-factor).
- The API port is **not published** to the host – only nginx on `:8080` is reachable.

- `npm ci` from a committed lockfile; the dependency tree is audited (**0 vulnerabilities** at the time of writing).

Live deployment hardening:

- **JWT verification instead of trust:** the API validates the Cognito ID token's RS256 signature against the pool's JWKS using `node:crypto` (no SDK), then checks `iss`, `aud / client_id`, `exp` and `token_use`. A forged, expired, wrong-pool or non-ID token is a `401`
- **Session expiry is enforced client- and server-side:** the guard treats an expired token as signed out, and the API would reject it anyway
- **No SSH on the EC2 host:** port 22 is not opened at all; the security group allows 8080 only from the AWS-managed CloudFront origin-facing prefix list, and administration happens through SSM Session Manager / Run Command
- **IMDSv2 required**, one-hop response limit, and an encrypted `gp3` root volume
- **No long-lived AWS credentials:** GitHub Actions assumes a role via OIDC that is scoped to this repository, this bucket, this distribution and this instance
- **Runtime secrets stay on the host:** the deployed stack's Cognito ids, CORS origin and database passwords are written into a root-only `.env` by the CloudFormation user data script – never in Git, never in an image, and the passwords are `NoEcho` parameters in the stack
- **CloudFront enforces HTTPS** for viewers, and the S3 bucket is private with a bucket policy that only permits that distribution (OAC)
- **One AWS permission, one ARN:** the instance role's inline policy allows `cognito-idp:AdminDeleteUser` on the GameHub user pool and nothing else – no `CognitoPowerUser`, no `*` resource
- **No third-party requests from the browser:** fonts are self-hosted (OFL), so loading a page discloses no visitor IP to Google or to an analytics/CDN provider. This is a German data-protection requirement in practice, not a performance preference

Data protection (GDPR)

- **Data minimisation by schema:** one `(user_sub, game)` row per player holding the best score – no play history, no email in the score table, no IP next to a score
- **Storage limitation:** access logs are retained for 7 days and then deleted automatically; the retention is set on the log groups in `infra/observability.yaml` and the reason is written down in the notice
- **Erasure (Art. 17):** `DELETE /api/me` removes the database rows and the Cognito identity. The order matters – rows first – because if the identity were deleted first, a failure would leave data nobody (including the user) could ever delete. A Cognito failure returns `502` with a truthful message
- **Access & portability (Art. 15, 20):** the account page lists the identity and every stored row, with a one-click JSON export
- **Transparency (Art. 13):** `frontend/privacy.html` is a bilingual inventory – item, purpose, legal basis, retention, processors (AWS, `eu-central-1`) and rights
- **Not stored on purpose:** passwords (Cognito holds a salted hash we never see), advertising identifiers, behavioural profiles, third-party analytics

Supply chain & pipeline

- **SBOM per build:** an SPDX JSON inventory (syft) is generated in CI and kept as a build artifact
- **Two vulnerability scans, two questions:** Trivy over the filesystem (source and dependencies) *and* over the built image (what the base image adds), plus `npm audit` – all failing the job on `CRITICAL / HIGH`

- **Secret scanning over history**, not just the diff: gitleaks runs against the full git history, because a secret that was committed and removed is still leaked
 - **Dependabot** for npm, GitHub Actions and base images: updates arrive as reviewable pull requests that must pass the same CI, with no auto-merge
 - **The deploy is gated on CI, not on the push**: `.github/workflows/deploy.yml` triggers on a *successful CI run* for a specific commit and checks out **that** commit, so a merge cannot reach production before the tests have judged it
 - **Post-deploy verification against the public URL**: the pipeline finishes by checking the homepage, all 12 catalogue entries through the API, game details by id, and that an unauthenticated `/api/me` returns `401` – the edge, not just the host
 - **Branch protection on `main`**: the four CI jobs are required status checks in strict mode, administrators are not exempt, force-pushes and branch deletion are disabled, and unresolved conversations block the merge. The gate is only a gate if the merge itself is gated
-

Testing Checklist

- ☒ Homepage loads successfully. *[smoke check]*
- ☐ Home and Games navigation works.
- ☐ Tournaments and Leaderboard links navigate to their intended sections.
- ☒ Games page loads the catalogue [12 seeded games]. *[browser-verified]*
- ☐ Category filters display the expected games.
- ☒ Backend liveness endpoint responds. *[smoke check]*
- ☒ Backend readiness endpoint reports database state. *[smoke check]*
- ☒ Games API returns database records. *[smoke check]*
- ☒ Nginx successfully proxies API requests. *[smoke check on :8080]*
- ☒ Unknown routes return clean JSON 404s. *[smoke check]*
- ☒ Backend port is not reachable from the host. *[verified: connection refused]*
- ☒ Database credentials remain untracked [Git **and** image builds].
- ☒ A clean rebuild + reseed works via `make bootstrap`. *[verified: full teardown → smoke passed]*
- ☒ All 12 catalogue entries open a real game. *[engines tested in CI + browser spot-checks]*
- ☒ Game engines survive hundreds of frames and reach their terminal states. *[21 frontend tests]*
- ☒ Every seeded game maps to an engine [no dead catalogue entries]. *[mapping contract test]*
- ☒ Cognito tokens are verified server-side; bad/expired/wrong-pool tokens get 401. *[12 auth tests]*
- ☒ Sign-in gate stays open [with a loud boot warning] when no pool is configured. *[auth test]*
- ☒ Live site deployed to S3 + CloudFront, API reachable through CloudFront `/api/*`. *[verified live]*
- ☒ Deployment is keyless: GitHub Actions assumes a role via OIDC. *[deploy workflow]*
- ☒ Structured JSON access logs reach CloudWatch Logs. *[verified with Logs Insights]*
- ☒ Log-derived metrics produce data points. *[verified with `get-metric-statistics`]*
- ☒ The synthetic check runs on schedule and its result is a metric, not just a log line. *[verified live]*
- ☒ Every alarm has a real state and a notification target. *[verified with `describe-alarms`]*
- ☒ The app runs on Kubernetes from the same images. *[verified on kind; smoke test passed on :8081]*

- ☒ Manifests are valid and the generated DB seed matches `db/init.sql`. *[kubecnform + drift check in CI]*
 - ☒ Argo CD reports the Application `Synced` / `Healthy` and reverts manual drift. *[verified on kind]*
 - ☒ Rolling updates never drop below the declared replica count. *[`maxUnavailable: 0`]*
 - ☒ A player's scores are stored server-side, one row per game, best only. *[gdpr test: upsert semantics]*
 - ☒ A player can retrieve every row stored about them. *[gdpr test: `/api/scores/me`]*
 - ☒ One player's query returns nobody else's rows. *[gdpr test: row isolation]*
 - ☒ Account deletion removes the DB rows **and** the Cognito identity. *[gdpr test: SigV4 `AdminDeleteUser` asserted]*
 - ☒ A Cognito failure still erases the data and reports a partial result `[ 502 ]`. *[gdpr test]*
 - ☒ The identity cannot be deleted by a request that has no pool configured. *[open-mode test: `501`]*
 - ☒ Signatures are real: session tokens are signed, and a changed body changes the signature. *[10 signer tests]*
 - ☒ Migrations apply once, are idempotent, and are refused if rewritten. *[CI runs the runner twice]*
 - ☒ The built image is scanned, not just the source tree. *[Trivy image scan in CI]*
 - ☒ A dependency inventory (SBOM) is produced on every build. *[syft, SPDX JSON artifact]*
 - ☒ Production only receives a commit that CI passed, and the deploy is smoke-tested at the edge. *[deploy workflow `workflow_run` + `verify`]*
-

Skills Demonstrated

- Frontend and backend integration
- REST API development
- Relational database connectivity
- MySQL schema bootstrap and seeding
- Docker image creation
- Multi-container application orchestration
- Docker network communication
- Nginx reverse proxy configuration
- Environment-based configuration
- Healthcheck-driven container startup [ordered dependency waits]
- Liveness vs readiness probe design
- Connection pooling
- Non-root container images with **no package manager in the runtime**, base images on the stable channel with build-time security patches, audited lockfiles
- One-command bootstrap + smoke testing
- Automated testing [unit + integration] with coverage gates
- CI/CD pipelines [GitHub Actions]: lint → test → build → security scan
- Dependency and image security scanning [Trivy, npm audit, gitleaks]
- GDPR-conscious frontend: self-hosted assets, zero third-party requests
- Canvas-based UI animation and game development [6 engines, game-loop design]

- Testing graphics code without a browser (stubbed canvas, frame-driven tests)
 - Canvas-based UI animation with accessibility (reduced-motion) support
 - Identity and access: Cognito user pools, ID tokens, JWKS-based JWT verification
 - Keyless CI/CD: GitHub Actions OIDC federation and least-privilege deploy roles
 - Infrastructure as code: five CloudFormation stacks with cross-stack outputs
 - Static hosting and CDN design: private S3 + CloudFront OAC, cache-behaviour split
 - Observability: structured JSON logs, log-derived metrics, metric math on a dashboard
 - Synthetic (black-box) availability monitoring with alarm semantics that treat silence as failure
 - Alert design: 8 actionable alarms, SNS notification path, `TreatMissingData` chosen per signal
 - SLOs and error budgets written before the incident, with a per-alarm runbook
 - Kubernetes: Deployments vs StatefulSets, named ports, probes, resource requests/limits
 - Container security context: capability drop lists chosen from evidence, seccomp, non-root
 - GitOps with Argo CD: automated sync, prune, selfHeal, drift as a bug to be reverted
 - Local cluster engineering: kind, side-loaded images, NodePort mapping, PVC scope
 - Host hardening: SSM-only administration, IMDSv2, prefix-list-restricted ingress
 - Data protection as code: GDPR art. 15/17/20 implemented as endpoints, not policy text
 - Erasure across two systems with a defined failure mode (partial deletion reported as partial)
 - AWS request signing by hand (SigV4) and instance-metadata credentials (IMDSv2), no SDK
 - IAM least privilege: one action, scoped to one resource ARN
 - Data minimisation in schema design (one row per player per game, pseudonymous key)
 - Database migrations with a checksum ledger; idempotency verified in CI
 - Supply chain: SBOM generation, image-level vulnerability scanning, Dependabot, secret scanning over history
 - CI-gated deployment: production receives only the commit CI passed, then a post-deploy smoke test at the edge
 - Git and GitHub workflows
 - Application troubleshooting
-

Roadmap

Phase	Focus	Status
0	Reproducible local foundation (compose, seed, healthchecks, <code>make bootstrap</code> , smoke tests)	✅ 2026-10-18
1	Automated tests + GitHub Actions CI (lint, coverage gate, Trivy, gitleaks)	✅ 2026-10-18
2	Game-y UI: live animated background, self-hosted game fonts, arcade styling	✅ 2026-10-18
3	Real playable games – every catalogue entry opens a game	✅ 2026-10-19
4	Auth & live: Cognito (sign-in gate), S3 + CloudFront, EC2 backend, CloudFormation + OIDC CD	✅ 2026-10-19
5	Observability: structured logs, metrics from logs, synthetic check, 8 alarms, dashboard, SLOs, runbook	✅ 2026-10-19
6	Local Kubernetes (kind) + Argo CD GitOps	✅ 2026-10-19
7	Security & GDPR hardening: account deletion (art. 17) across DB + Cognito, SigV4 without an SDK, migrations with a checksum ledger, SBOM + image scanning, CI-gated deploys	✅ 2026-10-19
8	/learn documentation site + PDF export + load-test report	✅ 2026-10-19

Author

Cijo Stany John

GameHub is an evolving learning and portfolio project built to practise AWS cloud architecture, backend development, containerization, and DevOps deployment workflows.

Built with curiosity, containers, and a passion for gaming. 🎮

Step 0 – Reproducible Local Foundation

Verified by: `scripts/smoke.sh` [5 checks] · full `make clean` → `make bootstrap` teardown-and-rebuild · browser check [12 DB-backed cards]

What was built

The stack went from **not runnable at all** [dead AWS RDS endpoint, no local database, Docker runtime down] to `git clone && make bootstrap` giving a fully working, seeded application with **zero configuration**.

Piece	Decision	Why not the alternative
MySQL 8.4 container + <code>db/init.sql</code>	Auto-seeds schema + 12 games on first boot of an empty volume	<i>Cloud DB now</i> → needs an account, costs money, breaks fresh clones; <i>no DB</i> → the original state, catalogue dead
Connection pool [<code>mysql2/promise</code>]	One shared pool, <code>connectionLimit: 10</code>	<i>Connection per request</i> [what existed] → TCP+TLS handshake on every request, exhausts MySQL <code>max_connections</code> under load
Liveness vs readiness split	<code>/api/health</code> never touches the DB; <code>/api/ready</code> does	<i>Health checks the DB</i> [what existed] → a database blip makes the orchestrator think the app is dead → crash-looping containers
Healthchecks + ordered startup in compose	<code>db</code> healthy → <code>backend</code> healthy → <code>frontend</code> starts	<code>depends_on</code> <i>without conditions</i> → app starts before MySQL accepts connections → first queries fail
No published API port	Only nginx <code>:8080</code> reaches the host	<i>Publishing <code>:3000</code></i> → the API bypasses the proxy layer on every developer machine
Pinned images + <code>npm ci</code> + non-root	<code>node:22-alpine</code> , <code>nginx:1.29-alpine</code> , <code>mysql:8.4</code> , uid 1000	<code>node:alpine</code> + <code>npm install</code> → unreproducible builds, root processes

The bugs we hit

1. **Critical npm advisory** [`proxy-addr`, IP spoofing via IPv4-mapped IPv6] surfaced by `npm audit` during the build → fixed with `npm audit fix`, lockfile updated, image rebuilt, 0 vulnerabilities since.
2. **A README claim was false:** it said "AWS RDS integration" while the configured endpoint did not even resolve [NXDOMAIN]. The honest fix – a local database plus an honest status table – beat any workaround.

Concepts to understand cold

- Why a **pool** beats per-request connections [and what `connectionLimit` protects]
- **Liveness vs readiness:** which probe answers "should I restart you?" vs "should I send you traffic?" – and why mixing them causes crash loops
- **Layer caching:** dependency layer [`package.json` + lockfile] before source so code edits don't re-download packages
- What `docker-entrypoint-initdb.d` does [and why it only runs on an *empty* volume]

Interview questions this unlocks

1. "Your service is running but `/api/ready` returns 503 – what's happening and what do you do?"
2. "How do you make an application reproducible for a new developer?"
3. "Why is your backend port not exposed on the host?"
4. "The database container was recreated and your data is gone – why, and how do you prevent it?" (named volume; `make db-reset` is the *intentional* wipe)

Step 1 – Automated Tests + GitHub Actions CI (2026-10-18)

Verified by: 14/14 tests green · coverage gate [95% lines / 81% branches / 90% functions, gate at 80] · lint clean · CI badge on `main`

What was built

Layer	Tooling	Scope
Unit suite	<code>node:test</code> [built-in – zero test dependencies]	Liveness, 404 JSON, CORS allow/deny, header hardening, malformed JSON → 400, DB-outage behaviour
Integration suite	<code>node:test</code> + real MySQL	Readiness, catalogue shape/ordering/uniqueness, category contract with the UI
Gate	<code>--test-coverage--lines/branches/functions=80</code>	CI fails under 80% – the number is enforced, not decorative
Lint	ESLint 9 flat config	<code>eslint .</code> is a CI step, not advice
CI	GitHub Actions, 3 jobs	① lint+test+coverage vs a MySQL service container ② real-image stack smoke test ③ security : gitleaks / Trivy / npm audit

Design decisions

- **Integration skips loudly instead of failing** when MySQL is unreachable – a fresh clone with Docker off still gets green unit tests, and the console says exactly why integration was skipped.
- `/api/health` **survives a database outage by test**, not by claim: the suite loads the app pointed at a *closed port* and asserts liveness=200, readiness=503. This is the test that prevents the crash-loop anti-pattern from regressing.
- **No supertest/jest**: `node:test` + built-in `fetch` covers everything here – fewer dependencies is itself a supply-chain decision (npm audit stays 0).
- MySQL published on `127.0.0.1` **only** so host-run integration tests work; CI uses a service container instead of the developer's database.

The bugs we hit

1. **Test process hung forever** after passing – the MySQL pool kept the event loop alive. Fix: `after()` now closes keep-alive sockets [`server.closeAllConnections()`] and `await pool.end()`.
2. **Coverage gate failed at 78.6% branches** – the honest fix was *new tests* (DB-error 500 path, TLS config branch, JSON parse failure), not a lower threshold. The gate also exposed a real defect: malformed JSON returned 500; it now returns 400 (client error).

Concepts to understand cold

- **Service containers** in GitHub Actions vs local Docker – same MySQL image, different wiring

- What a **coverage gate** does and why coverage $\neq$ correctness (we test *behaviour*: CORS refusal, outage semantics, error non-leakage)
- **Test isolation:** `node --test` runs each file in its own process – why that matters when tests mutate `process.env` and `require.cache`

Interview questions this unlocks

1. "Your CI coverage gate fails at 78% – do you lower the threshold?"
2. "How would you test that your app behaves correctly during a database outage?"
3. "What's the difference between a unit test and an integration test in your repo, and when does each run?"
4. "Name two things your security job would catch before a human reviews the PR."

Step 2 – Game-y UI: Live Background + Self-Hosted Arcade Fonts (2026-10-18)

Verified by: browser inspection (0 third-party requests, all three font faces active, canvas layer present) · `bg.js` render logic executed against a stubbed DOM (118 draw calls/frame; zero frames under `prefers-reduced-motion`) · `make smoke` 5/5 · CI full-stack job rebuilds the immutable frontend image

What was built

Layer	Implementation	Why this and not the alternative
Fonts	Press Start 2P (arcade accents) + Chakra Petch (headings/body), self-hosted woff2 with OFL licenses in-repo	<i>Google Fonts CDN</i> (what existed – an <code>@import</code> in <code>style.css</code>) → every visitor's IP is disclosed to Google on page load; German courts ruled this unlawful without consent (GDPR/§25 TTDSG). Kills Lighthouse third-party requests too
Live background	One <code><canvas></code> : 3-layer parallax starfield (pixel squares), rising neon pixel particles, rare shooting stars – ~118 draw calls/frame, zero dependencies	<i>CSS-only gradients</i> (what existed – static) → user explicitly asked for a moving, "games time" background; <i>a particle library</i> → supply-chain weight for 150 lines of canvas we control
Always-moving layers	Grid scrolls upward (CSS <code>background-position</code> keyframes), one blurred neon orb drifts (<code>body::after</code> , fixed layer)	Cheap GPU-friendly transforms; content stays above (negative z-index layers)
Accessibility	<code>prefers-reduced-motion</code> : canvas never created, grid/orb animations off	A moving background must not violate user motion preferences
Arcade controls	Press Start 2P on buttons/filters/logo/badges (kept ≤10px – the font is wide), Chakra Petch 700 on headings	Full-page Press Start 2P would destroy readability and overflow cards

The bugs we hit

- The edited CSS never appeared in the browser while `curl` showed it served.** Root cause: `Last-Modified: Sep 16` on the old file gave the browser *heuristic freshness* (10% of file age ≈ 3 days) – it never revalidated. Lesson (and production truth): the frontend ships as an **immutable Docker image**, so any UI change requires `docker-compose build frontend` – bind-mounting for "quick edits" would fork local behaviour away from production.
- Canvas "wasn't animating" in tooling.** Root cause: `requestAnimationFrame` is paused in hidden windows. The render loop was verified by executing `bg.js` against a stubbed DOM in Node instead – an executable proof that doesn't depend on a visible desktop.

Concepts to understand cold

- GDPR and third-party assets:** why self-hosting fonts is a legal requirement in Germany, not a preference – and why `font-display: swap` keeps LCP intact

- **Heuristic caching:** no `Cache-Control` header → browser derives freshness from `Last-Modified` age; the debugging workflow when "my change isn't there"
- **Immutable artifacts:** same image in dev, CI, and prod — changes mean a rebuild, and that's the point
- **Canvas vs CSS** for continuous animation: when `requestAnimationFrame` wins (many moving objects) and when transforms/keyframes win (few large shapes)

Interview questions this unlocks

1. *"A style change works in curl but not in the browser — walk me through how you'd diagnose it."*
2. *"Your Lighthouse audit flags `fonts.googleapis.com` — is that a performance or a compliance problem in Germany? How do you fix both?"*
3. *"How do you make a continuously animated background accessible and battery-friendly?"*
4. *"Why can't developers just edit files inside the running container?"*

Step 3 — Real Playable Games (2026-10-19)

Verified by: 40 automated tests (19 backend + 21 frontend) · engine tests run the real game loops in Node · browser spot-checks on 4 engines + the 404 path · catalogue→runner click-through · `make smoke` 5/5 · CI full-stack job

What was built

Every one of the 12 catalogue entries now opens an actual, playable game. No placeholders, no alerts — a `play.html?id=<n>` runner that fetches the title, dispatches it to a real engine, and drives a READY → PLAYING → GAME OVER state machine with HUD, per-title high scores (localStorage) and controls help.

Piece	Implementation	Why not the alternative
6 engines (<code>engines.js</code>)	snake · breakout · dodge (survival racing) · memory · reflex · pong — all plain canvas 2D, zero dependencies	<i>12 bespoke games</i> → weeks of content work with no new DevOps signal; <i>one engine themed 12 ways</i> → dishonest ("every game is snake"); 6 engines × 2 titles is real variety at portfolio scale
Title → engine mapping	<code>ENGINE_MAP</code> in <code>play.js</code> keyed by game name (a uniqueness contract already enforced by a backend integration test)	An <i>engine DB column</i> → schema migration + <code>make db-reset</code> for a frontend concern; the name-uniqueness test already guards the invariant
Coverage contract test	<code>mapping.test.js</code> parses the DB seed and <code>play.js</code> and fails CI if any seeded title lacks an engine (or vice versa)	<i>Manual checking</i> → the promise "every game is playable" would silently rot the day someone adds a seed row
Engine tests	The real engines run in Node against a stubbed canvas/DOM (<code>vm</code> + Proxy ctx), driven frame-by-frame	<i>Browser automation</i> → a hidden window never fires <code>requestAnimationFrame</code> ; headless pixels are not behaviour. Frame-driving asserts real terminal paths (snake hits the wall, breakout runs out of lives, pong reaches 5)
GET <code>/api/games/:id</code>	Single-title endpoint with 404 for unknown and non-numeric ids (no crash, no leak)	<i>Fetching the whole list</i> → the runner only needs one title; the endpoint is also the natural REST shape a reviewer looks for

The bugs we hit

- The engine tests found a real gameplay stall.** In breakout, a dead-centre paddle hit computes `vx = 0` → the ball bounces vertically forever on a stationary paddle and the game can never end. Fixed with a minimum horizontal component (`speed * 0.25`). A test that couldn't fail this way would have been a decoration.
- 413 was missing.** Oversized JSON bodies (body-parser's `entity.too.large`) fell through to the central 500 handler. Now: 413 with a clean body — and the fall-through path itself is covered by a test.
- The running container served the old API.** `/api/games/1` returned the 404 handler because the backend container still ran the Phase 0 image; local `curl` against the *new* code hid it. Lesson: images are immutable — a route is not deployed until the image is rebuilt.

Concepts to understand cold

- **Game loop fundamentals:** fixed-step update + render, why `dt` clamping matters, and why input should be queued (snake buffers direction changes so a fast double-tap can't reverse the snake into itself)
- **Testing graphics code without pixels:** stub the context, drive frames, assert *state transitions* (score, lives, game over) — behaviour over pixels
- **Contract tests as product promises:** "every catalogue entry is playable" converted into a failing build when violated
- **REST resource design:** list vs detail, idempotent reads, honest 404s for malformed identifiers

Interview questions this unlocks

1. "How do you test canvas game logic in CI without a browser?"
2. "Your test suite can't fail — how do you know it's actually testing anything?" (the breakout stall is the answer)
3. "Someone adds a 13th game to the seed — what stops the catalogue from shipping a dead entry?"
4. "Why did `curl` succeed while the browser got a 404?" (immutable images)

Step 4 — Real Auth + Live on AWS (2026-10-19)

Verified by: 64 automated tests (31 backend + 33 frontend) · Cognito signing key pair exercised with real RSA signatures · live site served from CloudFront · `/api/*` answering through the CloudFront origin · a confirmed test user signing in and calling `/api/me` with a real ID token

What was built

Players now sign up and sign in before they reach the games, and the whole thing runs on AWS as code — no console clicking, no SSH, no stored access keys.

Piece	Implementation	Why not the alternative
Sign-in gate [<code>frontend/auth.js</code>]	Raw Cognito REST calls: <code>SignUp</code> , <code>ConfirmSignUp</code> , <code>InitiateAuth</code> [<code>USER_PASSWORD_AUTH</code>]	<i>Amplify or the AWS SDK</i> → hundreds of KB of JavaScript for three endpoints, plus a bundler this project intentionally does not have
Token verification [<code>backend/auth.js</code>]	RS256 checked against the pool JWKS with <code>node:crypto</code> ; <code>iss</code> , <code>aud</code> / <code>client_id</code> , <code>exp</code> , <code>token_use</code> all validated	<code>jsonwebtoken</code> + <code>jwt-rsa</code> → two dependencies to do what <code>createPublicKey({format:"jwk"})</code> already does; <i>trusting the frontend</i> → not security
Enforcement model	Middleware on <code>/api/*</code> except <code>/api/health</code> and <code>/api/ready</code> , active only when the pool is configured	<i>Always-on</i> → local development and CI would break or need fake pools; <i>client-only guard</i> → trivially bypassed with <code>curl</code>
Frontend hosting [<code>infra/site.yaml</code>]	Private S3 bucket + CloudFront with Origin Access Control; <code>/api/*</code> behaviour with caching disabled	<i>Public bucket</i> → everyone can read and write objects; <i>caching the API</i> → stale data and cross-user cache confusion
API host [<code>infra/backend.yaml</code>]	One EC2 instance running the same <code>docker-compose.yaml</code> as local development, with a 2 GB swap file	<i>ECS/Fargate + RDS</i> → more moving parts and cost for a portfolio demo; <i>Lambda</i> → the MySQL container would have to go, which is the opposite of "the deployed thing is the local thing"
Access model	No SSH. Ingress on 8080 only from the AWS-managed CloudFront origin-facing prefix list; admin via SSM; IMDSv2 required	<code>0.0.0.0/0</code> on 22 + a key pair → the most common way a portfolio EC2 becomes someone else's crypto miner
Deploys [<code>github-oidc.yaml</code> + <code>deploy.yaml</code>]	GitHub OIDC → role scoped to this repo, bucket, distribution and instance; site published by <code>s3 sync</code> , API host rolled out by an SSM command	<i>Long-lived IAM keys in GitHub secrets</i> → a permanent credential to leak; <i>rebuilding the host per deploy</i> → slower, and the DB volume lives there

Things that had to be got right

1. **AutoVerifiedAttributes** + **email-only username**. New pools make MFA mandatory unless you opt out, and users sign in with an email address rather than a username. Both are explicit in the template.

2. **Cognito emits `client_id` on access tokens and `aud` on ID tokens.** The verifier accepts either, then rejects non-ID tokens outright – accepting an access token here would hand out a weaker identity contract.
3. **The probes must never 401.** `/api/health` and `/api/ready` are exempt from the gate; an orchestrator that cannot reach its own health endpoint kills a healthy service.
4. **CloudFront needs a domain name, not an IP, for a custom origin.** The backend stack outputs the instance's public DNS name while the elastic IP keeps it stable.
5. **A 1 GB instance running MySQL wants swap.** The user-data script creates a 2 GB swap file before starting the stack – cheap insurance against the OOM killer during a `docker compose build`.
6. **The artifact carries its config, the repo does not.** `config.js` ships with `authEnabled: false` for local development; `scripts/build-frontend.sh` writes the real pool ids into a git-ignored `build/` directory at deploy time.
7. **An edge invalidation is not a browser invalidation.** After a deploy the browser still ran the previous `script.js` from its week-long cache, which made a fixed landing page look broken. The build now stamps every asset URL with the deploy version [`script.js?v=<sha>`] while HTML stays on a one-minute cache – the standard two-tier answer to "how do users get the new version without a hard refresh?"

Concepts to understand cold

- **OIDC federation:** how a GitHub-issued JWT becomes temporary AWS credentials, and why the `sub` condition is what stops any other repository from assuming your deploy role
- **JWKS and key rotation:** why the verifier fetches public keys instead of a shared secret, and why the result is cached
- **Edge vs origin caching:** why static assets get a week and HTML gets a minute, and why `/api/*` must not be cached at all
- **Origin Access Control:** keeping an S3 bucket private while still serving it through a CDN
- **Blast-radius reduction:** no SSH, SSM-only admin, prefix-list ingress, IMDSv2

Interview questions this unlocks

1. "How does your API know a request comes from a signed-in user?" (signature, issuer, audience, expiry – and why client-side checks are not the boundary)
2. "How do your CI pipelines authenticate to AWS, and what happens if the repo is compromised?" (no keys; role scoped to repo, bucket, distribution, instance)
3. "Why is the S3 bucket private, and how does the CDN still read it?" (OAC + bucket policy condition on the distribution ARN)
4. "What is your EC2 host's attack surface?" (no SSH, one port, one prefix list, SSM only, IMDSv2)
5. "How do you deploy config that differs per environment without committing it?" (generate at build time into a git-ignored directory)

Step 5 – Observability (2026-10-19)

Verified by: the CloudWatch agent running on the live host, structured JSON nginx access logs arriving in `/gamehub/prod/nginx-access`, container stdout arriving in `/gamehub/prod/app`, `CWAgent` host metrics with the dimensions read back from `list-metrics`, all eight alarms reporting a real state (including the memory and disk alarms built from those dimensions), the synthetic canary invoked by hand and by schedule, and one dashboard [`gamehub-prod`] whose log widgets return rows.

What was built

A deployment you cannot see is a deployment you cannot run. This step makes the live system observable from the outside in and the inside out.

Piece	Implementation	Why not the alternative
Structured access logs [frontend/nginx.conf]	log_format ... escape=json with status , uri , request_time , upstream_time	Parsing the default combined log with regexes in a metric filter → every rule becomes a cubicle joke; JSON metric filters are a first-class CloudWatch feature
Shipping logs + host metrics [infra/backend.yaml]	CloudWatch agent installed by user-data, config file written inline, logs → CloudWatch Logs, mem / disk / swap → the CWAgent namespace	Container Insights / an APM vendor → an agent you must pay per host for, when four metrics and two log files answer every question this project has
Retention [observability.yaml]	Log groups created by template with 7-day retention	Letting the agent create them → groups that never expire and quietly bill forever is the classic first-week-of-AWS mistake
Metrics from logs [AWS::Logs::MetricFilter]	Requests , ServerErrors , RequestTimeSeconds , ApiRequestTimeSeconds derived from the JSON log lines	Instrumenting the app with an EMF library → more code in the request path for numbers nginx already knows
External availability check	EventBridge rate(5 minutes) → a 20-line Lambda that fetches / and /api/health and prints one JSON line per check; a metric filter turns that line into SyntheticSuccess	CloudWatch Synthetics → the managed canary costs about \$10/month at a 5-minute interval, more than the whole project; printing a line and filtering it is free
Alarms [observability.yaml]	8 alarms: availability (synthetic), 5xx count, API p99 latency, instance status, CPU, CPU credits, memory, disk – all publishing to one SNS topic	A single "something is wrong" alarm → an alert nobody can act on is an alert everybody mutes
One dashboard	Availability, requests vs errors, latency percentiles, host pressure, and two live log tables (recent 5xx, backend log tail)	Per-service dashboards → this service has one host; the dashboard fits on a screen
Written response [docs/slo.md , docs/runbook.md]	Objectives with error budgets, measurement, and one runbook section per alarm containing the exact commands	"Investigate and fix" → the runbook's value is that it was written before the incident, by someone who was not panicking

Things that had to be got right

- TreatMissingData is a decision, not a default.** For the host metrics, missing data is notBreaching – a stopped agent should not page anyone. For the synthetic check it is breaching : no data means the checker itself is broken, and an availability monitor that fails silently is worse than none.
- The agent must not create the log groups.** If the agent starts first, the groups exist without retention and CloudFormation's AWS::Logs::LogGroup fails with AlreadyExists on the next deploy. The template owns the groups; the agent only writes into them.
- A metric filter is not a log parser you can debug in place.** Pattern syntax errors produce an alarm that never fires – no warning, no log entry. The check that matters is a **manual canary invocation followed by get-metric-statistics** , proving a data point exists, not that the template was accepted.

4. **Metric dimensions must match exactly.** `CWAgent` attaches `InstanceId`, and the disk plugin adds `path`, `device` and `fstype` on top. An alarm whose dimension set is one key short sits in `INSUFFICIENT_DATA` forever. The dimensions in this stack were read back from `list-metrics` on the running host, then written into the template – verified, not guessed.
5. **The synthetic check measures the path a user takes.** It calls the CloudFront URL, not localhost – otherwise it proves the host is alive while the CDN is broken, which is exactly the failure it is meant to catch.
6. **Alarms without a subscriber are half a feature.** The topic is created and exported; subscribing is one `aws sns subscribe` because the account's own contact address cannot be read by an alarm template. Documented in `docs/runbook.md` §8 rather than pretended away.
7. **Latency is measured where the user waits.** `$request_time` includes the upstream Express call; `$upstream_time` alone would hide the time nginx spends. Having both on the log line is what lets the runbook split "slow app" from "slow edge" with one query.
8. **A metric filter publishes its default value for every event that does not match.** The canary alarm was built on `Average` of a metric emitted from a Lambda log group, and it sat in `ALARM` with an average of `0.4` while every check was passing: the group also contains `START`, `END` and `REPORT` lines, each of which contributed a zero. The fix is not a different threshold, it is a different *statistic* – one JSON line per run, summed, so the zeros add nothing. The same trap is why numeric-extraction filters [`$.request_time`] must only be pointed at log groups where every event matches; the nginx group qualifies because only `/api/*` traffic reaches the origin, and the S3-served page requests never enter it.
9. **The agent has to be root to read Docker's container logs.** `/var/lib/docker/containers/*/*-json.log` is `0640 root:root` inside a `0710` directory. Running the agent as its own `cwagent` user collected host metrics perfectly and shipped *nothing* from the containers – a silent half-failure, because the metrics half looked healthy. The log group stayed empty until `run_as_user` was set to `root`, with the reason written next to it.
10. **A policy that is only in the template does not exist on a running host.** The instance role gained `CloudWatchAgentServerPolicy` in the template, but the live host was created before that, so the agent started and was immediately refused with `AccessDeniedException: ... not authorized to perform: logs:PutLogEvents`. IaC is a description of the desired state; the running instance remains the old state until it is redeployed or the change is applied. (Attaching the policy by hand made reality match the template again – the template was already correct.)

Concepts to understand cold

- **The four golden signals** – latency, traffic, errors, saturation – and why this dashboard is organised as exactly those four plus availability
- **Metric filters and JSON patterns** – how a log line becomes a time series, including extracting a *numeric value* from a field [`$.request_time`]
- **Extended statistics** – why the `p99` alarm exists next to the average, and why an average latency is the metric that hides the outage
- **Error budgets** – the shift from "uptime is good" to "the budget decides what we work on next", and what burning 3 h 39 m of a 30-day window means
- **Pull vs push monitoring** – why the synthetic check lives outside the system it watches, in a different service, in a different failure domain
- **Retention and cost of observability** – log ingestion is the subscription trap of cloud monitoring; retention is an architectural choice

Interview questions this unlocks

1. *"How do you know your service is up right now?"* (external synthetic check on the public URL, every 5 minutes, with `TreatMissingData: breaching` – and what that costs, which is nothing)
2. *"Walk me through an incident: the p99 latency alarm fired. What do you do?"* (runbook §3: split `request_time` vs `upstream_time`, then CPU credits, then MySQL swap)
3. *"How do you get metrics out of a system you did not instrument?"* (structured JSON logs → metric filters, versus adding an SDK or an EMF library)
4. *"Your alarm fired at 03:00. Was it a false positive, and how would you know?"* (dimension sets, `TreatMissingData`, evaluation periods – and the manual canary invocation that proves the wiring)
5. *"What does your error budget buy you, and who decided the target?"* (99.5 % on a single-AZ t3.micro is the honest ceiling; the budget decides whether the next sprint is features or reliability)

Step 6 — Local Kubernetes (kind) + Argo CD

✓ (2026-10-19)

Verified by: a real single-node kind cluster (4 CPU / 6 GiB Colima VM) with the app running from the same images as the Compose stack — `scripts/smoke.sh` passing against the cluster's NodePort on `:8081` (12 games served from the StatefulSet through the nginx proxy, and the documentation landing page at `/learn/`, asserted by marker rather than by status) — and Argo CD reporting the `gamehub` Application as `Synced / Healthy` at the merged revision, with a manual `kubect1 scale --replicas=1` reverted to the declared 2 replicas by `selfHeal` within one poll. Re-run in full after the dependency roll-up, which is how the stale-image trap in item 11 was found.

What was built

The Compose stack answers "can you run this?"; this step answers "can you run it on Kubernetes, and can a controller keep it that way?".

Piece	Implementation	Why not the alternative
Cluster [<code>k8s/kind-config.yaml</code>]	kind, one node, node port 30080 published on host 8081	<i>minikube</i> → a VM plus a driver argument on every laptop; <i>EKS</i> → real money and real waiting for a portfolio demo; <i>Docker Desktop's Kubernetes</i> → not available with Colima
Workloads [<code>k8s/*.yaml</code>]	<code>Deployment</code> for frontend (2 replicas) and backend (2 replicas, rolling update with <code>maxUnavailable: 0</code>), <code>StatefulSet</code> + PVC for MySQL	<i>A Deployment for MySQL</i> → two pods fighting over one data directory, and no stable identity for the client to resolve; <i>Helm</i> → a templating layer for three objects, and templating hides the fields this project exists to learn
Images	The same tags Compose builds [<code>gamehub-frontend:local</code> , <code>gamehub-backend:local</code>], side-loaded with <code>kind load docker-image</code> , referenced with <code>imagePullPolicy: Never</code>	<i>Pushing to ECR for a local cluster</i> → a registry round trip per change with nothing to show for it; <i>Building inside the cluster</i> → a Docker-in-Docker story nobody wants on a laptop
Service names	The backend Service is called <code>backend</code> , not <code>gamehub-backend</code>	The nginx config baked into the frontend image proxies to <code>http://backend:3000</code> . The image is identical in Compose and Kubernetes, so the Service name is part of the image's contract, not a naming preference
Probes	Liveness on <code>/api/health</code> (no DB), readiness on <code>/api/ready</code> (DB)	A liveness probe that checks the database turns a DB outage into a cluster-wide restart storm – the classic Kubernetes own-goal
Seed data	<code>db/init.sql</code> rendered into a ConfigMap by <code>scripts/build-k8s-dbinit.sh</code> , mounted at <code>/docker-entrypoint-initdb.d</code> , with a CI drift check	<i>Copying the SQL by hand</i> → two schemas that disagree the first time someone edits one; <i>an init Job</i> → more moving parts than the entrypoint contract MySQL already provides
GitOps [<code>k8s/argocd/application.yaml</code>]	Argo CD pulls <code>k8s/</code> from <code>main</code> , <code>automated</code> + <code>prune</code> + <code>selfHeal</code>	<code>kubectl apply</code> in a pipeline → the cluster state is whatever the last successful build left behind, with no answer to "what is deployed?" or "who edited this?"
Verification	The same <code>scripts/smoke.sh</code> as Compose, pointed at the NodePort	A separate "Kubernetes smoke test" → two definitions of working, which is how environments drift

Things that had to be got right

- log_format** is an http-level directive. The nginx config is a `server` block placed in `conf.d/`, so the structured log format had to move into its own file [`frontend/nginx-log-format.conf`, copied in as `00-log-format.conf`] or nginx exits with `"log_format" directive is not allowed here`. Found by the container restarting instead of serving – the failure was loud and immediate, which is the good kind.
- Ports belong to a machine, not to a plan.** kind's NodePort mapping wanted host 8080, which this Mac already uses (a Compose frontend *and* an unrelated SSH tunnel). The cluster now publishes on 8081, so

Compose and Kubernetes can run at the same time – and the smoke test takes the URL as an argument.

3. **prune and selfHeal are the whole argument for GitOps.** Without them, Argo CD is an expensive `kubectl apply` that never re-reconciles. With them, scaling a Deployment by hand is a *bug that gets fixed within a minute* – which is exactly what the verification demonstrates.
4. **A PVC on kind is a PV on a container filesystem.** It works, it persists across pod restarts, and it dies with the cluster. That is the correct scope for a learning environment; the AWS stack keeps its data on an EBS-backed volume with a snapshot schedule, and the two are not the same promise.
5. **imagePullPolicy: Never is a deliberate local choice.** It fails fast and loudly if the image was not side-loaded, instead of silently pulling a stale `latest` from a public registry that has nothing to do with this project.
6. **The seed ConfigMap is generated, not hand-maintained.** `db/init.sql` stays the single source of truth, `scripts/build-k8s-dbinit.sh --check` runs in CI, and a change to the schema that skips the render fails the build.
7. **readOnlyRootFilesystem was left off on purpose.** The nginx image writes its PID file and cache at startup and Node may use `/tmp`. Turning it on needs emptyDir mounts and an unprivileged nginx image – a change worth making, but only alongside the evidence that it still serves, which is why the manifests say so in a comment rather than pretending.
8. **drop: ALL is the wrong answer for the stock nginx image.** The root master process legitimately needs `CHOWN`, `DAC_OVERRIDE`, `SETUID`, `SETGID` and `NET_BIND_SERVICE` before it hands off to the unprivileged worker. Dropping everything produced `CrashLoopBackOff` in both replicas and the error `chown("/var/cache/nginx/client_temp", 101) failed (1: Operation not permitted)` in the container log – the pod was hardened straight into a crash, which is exactly why the capability list is now written down with its justification.
9. **The Argo CD install needs --server-side.** `kubectl apply` of the release manifest failed with `The CustomResourceDefinition "applicationsets.argoproj.io" is invalid: metadata.annotations: Too long: may not be more than 262144 bytes` – client-side apply stores the previous configuration in an annotation, and this CRD's own annotations blow past the limit. Server-side apply stores nothing in the object.
10. **read plus set -e is a trap in a wait loop.** `read -r a b < <(cmd)` returns non-zero when `cmd` produces no output, and under `set -eu` that ends the whole script – silently, on the first poll, while the Application has no status yet. The wait loop now collects the output first and parses it defensively, which is why the script prints its progress lines instead of exiting after "waiting for the first sync".
11. **A mutable image tag plus imagePullPolicy: Never means the cluster can serve yesterday's bytes.** Re-verifying this phase after the dependency upgrade exposed it: `kind load` had pushed the new image into the node, the image on disk plainly contained the documentation, and `get /learn/` in the cluster answered with the SPA home page. Nothing had restarted the Pods – the tag `[ :local ]` had not changed, so `kubectl apply` saw no spec difference and did nothing. `scripts/kind-up.sh` now restarts the workloads after loading them, which is the step that actually swaps the code in. The lasting lesson is about the *check*, though: the old smoke test only looked at status codes, and nginx's `try_files ... /index.html` fallback answers **200 with the wrong page** – the same failure mode as the CloudFront directory-URL bug (ADR-017), caught the same way the second time only because the assertion became a marker `[ readme.html ]` instead of a status. `scripts/smoke.sh` now asserts it, so both the Compose and the Kubernetes path would fail loudly on a stale image.
12. **The image carries the whole site, so local equals production.** The deployed artifact is `frontend/` plus `learn/` in one bucket; the frontend image now `COPY`s both, `.dockerignore` no longer excludes `learn/`, and the app's footer links to `/learn/`. nginx resolves a directory to `index.html` on its own, so the same

URL works in Compose and in kind without the edge function that CloudFront needs — a difference worth knowing when a link works locally and 404s in production, or the reverse.

Concepts to understand cold

- **Declarative reconciliation** — the desired state lives in Git, the controller compares it to the cluster and acts; nobody "deploys" anything
- **Probes and their consequences** — why liveness must not depend on a downstream service, and what `maxUnavailable: 0` buys during a rollout
- **Stateless vs stateful workloads** — which is why one of the three objects in this cluster is a StatefulSet and the other two are Deployments
- **Side-loading images** — how a node gets an image without a registry, and why a local cluster is the one place that is the right answer
- **Service DNS as an interface** — `backend:3000` as a contract an immutable image depends on
- **Drift** — what `selfHeal` actually detects and reverts, and what it cannot see (a changed ConfigMap value is drift; a changed image *inside* a tag is not)

Interview questions this unlocks

1. *"How would you deploy this to Kubernetes, and what breaks first?"* (image availability — hence `imagePullPolicy: Never` and side-loading; then the DB identity; then the probe that would otherwise restart every pod during a DB incident)
2. *"What is the difference between a Deployment and a StatefulSet, and when did you actually need one?"* (stable identity + stable storage; the MySQL pod)
3. *"Your pipeline says the deploy succeeded but the cluster is wrong. Why?"* (because `kubect1 apply` is a moment, not a state — Argo CD with `prune` and `selfHeal` is the answer, and it is observable)
4. *"How do you keep two environments' manifests from diverging?"* (one source of truth — the same image tags, the same ConfigMap source, and a CI check that fails when a generated file drifts)
5. *"What would you change for production?"* (an unprivileged nginx image with a read-only root filesystem, a real registry with digests instead of tags, a managed database, and resource limits sized from measurements rather than guesses)

Step 7 – Security hardening & GDPR

(2026-10-19)

Verified by: the CI job on this commit (51 backend tests, including a real RSA-signed JWT verified against a live JWKS and a hand-rolled SigV4 `AdminDeleteUser` call captured and asserted field by field against a local admin endpoint), a migration runner applying for real against MySQL 8.4 in CI **and in production** (the deploy log shows `0 already applied, 1 applied now` on the live database), and a live end-to-end erasure run by `scripts/verify-gdpr.sh` against `https://d2qf9zxehny6bk.cloudfront.net` (the distribution's name, the canonical name at the time; the site also answers on `gamehub.worldofvpc.online` – ADR-018): sign up → confirm → obtain an ID token → `POST /api/scores` [201] → `GET /api/scores/me` [1 row] → `DELETE /api/me` [204] → 0 rows remain **and** `admin-get-user` returns `UserNotFoundException`. That run is also what found a real production bug (item 13 below), which is the honest reason it is a script and not a checklist.

What was built

Security is not a feature you add at the end; it is a set of claims that are either true or false. This step is about making each claim checkable in the repository.

Piece	Implementation	Why not the alternative
No AWS SDK on the host [backend/cognito.js]	~120 lines: IMDSv2 credential fetch + SigV4 request signing for the single API call the app makes to AWS [AdminDeleteUser]	<i>Pulling in @aws-sdk/client-cognito-identity-provider</i> → tens of megabytes and a credential-provider chain to debug, to send one POST; the signing algorithm is the thing worth understanding
Credentials	IMDSv2 only: PUT /latest/api/token, then every metadata GET carries X-aws-ec2-metadata-token	<i>IMDSv1</i> → any SSRF in any container on the host becomes an instance-role theft; <i>long-lived access keys in .env</i> → a key on disk to rotate, leak and explain
Permission	An inline IAM policy on the instance role, scoped to one user-pool ARN [infra/backend.yaml]	<i>CognitoPowerUser</i> , or a * resource → the app could delete users in any pool of any account it can reach; least privilege here is one ARN
Score storage	One row per (user_sub, game) with score = GREATEST(score, VALUES(score)), keyed by the Cognito sub	<i>A row per play</i> → a behavioural history the app never needs and therefore should not hold; <i>keying by email</i> → copying a personal identifier into a second system for no reason
Erasure [DELETE /api/me]	Delete the DB rows first , then AdminDeleteUser; a Cognito failure answers 502 with a truthful message	<i>Delete the identity first</i> → the user can no longer authenticate, so a failed row deletion is unretryable and the data is orphaned forever
Rights built into the app [frontend/profile.html]	The account page is the access/portability answer (identity + every row, copyable as JSON) and the erasure control	<i>An email address in a README</i> → a GDPR right you have to be asked for is a right you have implemented on paper only
Notice [frontend/privacy.html]	Bilingual EN/DE notice: data inventory, purpose, legal basis, retention, processors, rights	<i>A link to AWS's privacy page</i> → answers the wrong question; the notice must say what this service does
Open mode	/api/* signed-in routes answer 501 when no Cognito pool is configured	<i>Returning 200 with an empty dataset</i> → local development silently exercises a path production never takes; <i>401 everywhere</i> → a contributor cannot tell "not configured" from "not allowed"
CI/CD [.github/workflows/]	Deploy runs only on a successful CI run for the exact commit , then smoke-tests the public URL; SBOM (syft), image scanning (Trivy), npm audit, Dependabot	<i>Deploy on push</i> → production can receive a commit that CI has not finished judging; <i>deploy and hope</i> → a deploy is not done when the command exits 0
Schema [db/migrations/, scripts/migrate.sh]	A ledger [schema_migrations] with a checksum per file; a rewritten migration is refused	<i>init.sql alone</i> → it runs once, on an empty volume, and never again — which is fine locally and wrong in production

Things that had to be got right

1. **AdminDeleteUser** is an admin API, and an admin API does not accept a user's token. The player's JWT proves *who is asking*; it cannot authorize a pool-wide operation. The backend therefore has to sign its own

AWS request – which is why this step contains a SigV4 implementation, and why the test suite asserts the signature rather than trusting it.

2. **SigV4 signs almost everything except the two things you expect.** The canonical request covers the method, the URI, the query string, the canonical headers, the *signed headers list* and the payload hash. Miss the payload hash and the signature is still "valid" to a naive check while binding nothing – the test that changes the request body and asserts the signature changes is the one that caught this class of mistake.
3. **Temporary credentials must be signed.** When the credentials come from IMDS, the session token is a *header that is part of the signature* [`x-amz-security-token`]. Adding it after signing produces an error that reads like a permissions problem and is not one.
4. **IMDSv2 is a two-step protocol, and that is the point.** `PUT` for a token, then attach it to every `GET`. The token is what makes an SSRF from inside a container useless – which is why the host is configured for token-required metadata and why the tests run the whole dance against a local stand-in server rather than a mock.
5. **The order of deletion is the design.** Data rows first, identity second. Delete the identity first and a failure leaves data that no one – including the user – can ever delete, because the user no longer exists to authenticate. The `502` message is deliberately blunt: *"Your game data was deleted, but the sign-in account could not be removed."* A half-finished erasure must never be reported as a completed one.
6. **A `sub` is pseudonymous, not anonymous.** The database holds no email address, but `sub` is still personal data under the GDPR. That is why the privacy notice lists it, and why it is deleted along with everything else.
7. **Request logs are personal data.** The nginx/CloudWatch access logs contain IP addresses, so the 7-day retention is a data-protection decision with a legal basis (legitimate interest), not just a cost setting. It is written down in the notice, in the same table as the email address.
8. **Self-hosted fonts are a German legal requirement in practice.** Loading `fonts.googleapis.com` transmits every visitor's IP to Google; German courts have held that this needs consent. The fonts here are self-hosted OFL files in `frontend/fonts/`, so the page makes no third-party request at all.
9. **`db/init.sql` only runs on an empty data volume.** That is a real property of the official MySQL image, and it means the file that documents the schema is not the file that updates a live one. Hence the migration runner – with checksums, because a migration that silently changed after being applied is worse than no migration ledger at all.
10. **A deploy is finished when the public URL says so.** The `verify` job runs after both deploys and checks the *edge* (CloudFront), not the host: real homepage markup, all 12 catalogue entries from the API, game details by id, and – the security assertion – that an unauthenticated `/api/me` returns `401`. A crashed container, a stale origin or a bad certificate fails this job.
11. **The security scans are in CI, so they are not optional.** Gitleaks over the full git history (not just the diff – a secret that was committed and removed is still leaked), Trivy against the *filesystem* and the *built image* (a package can be added by the base image), `npm audit`, an SPDX SBOM kept as a build artifact, and Dependabot for npm, Actions and base images.
12. **Branch protection makes the gate real.** The deploy is CI-gated, but CI is only a gate if the merge itself is gated. On `main`: the four CI jobs are required status checks with *strict* mode (the branch must be up to date), administrators are **not** exempt, force-pushes and deletion are disabled, and unresolved review conversations block the merge. A required human approval is deliberately absent – this is a one-person repository, and a rule that GitHub refuses to satisfy (you cannot approve your own pull request) is a rule that gets disabled the first time it is inconvenient.
13. **A container is one hop further from the metadata service than the host.** With `HttpPutResponseHopLimit: 1` – the AWS default, and what this template originally declared – the token `PUT` from inside the application container is never answered. Measured on the live host: the same request

returns `HTTP 200` from the host shell and dies with `ETIMEDOUT` from inside the container. The application sees `fetch failed` (undici's generic wrapper), the handler's catch turns it into `502`, and because the connect hangs for longer than CloudFront's origin timeout, the *user* sees a `504` from the CDN. Three layers separate the symptom from the cause, which is exactly why the value is now `2` **in the template** with the reasoning next to it rather than in a runbook. The live erasure check found this on production after the code had already been merged and deployed – the strongest possible argument for the check being a script that anyone can run.

Concepts to understand cold

- **SigV4**: canonical request → string to sign → derived signing key [`kSecret` → `kDate` → `kRegion` → `kService` → `kSigning`] → `HMAC-SHA256`. Why the key is derived per region and per service.
- **IMDSv2**: the token `PUT`, the one-hour default TTL, why link-local `169.254.169.254` is reachable from the instance and nothing else, and why `HttpTokens: required` is a hardening control.
- **GDPR as engineering**: controller vs. processor, legal basis (contract vs. legitimate interest), data minimisation, storage limitation, and the four rights that have code behind them here (access, portability, erasure, objection).
- **Least privilege**: an IAM permission scoped to a resource ARN [`userpool/eu-central-1_...`] instead of an action wildcard.
- **Supply chain**: SBOM ≠ vulnerability scan ≠ dependency update. Three different questions, three different tools, all running on every change.

Interview questions this unlocks

1. *Where does the authorisation decision actually happen when a user deletes their account, and what would break if the token were the only credential?*
2. *What is the failure mode of deleting the identity before the data, and how does your API report a partial erasure?*
3. *How would an SSRF in the frontend container escalate, and what stops it?*
4. *Your database is pseudonymous, not anonymous. Is that enough for the GDPR? Justify it.*
5. *Why is a migration checksum worth the complexity?*
6. *What does the deploy pipeline prove before traffic reaches a user?*
7. *Which of the four GDPR rights would you implement first in a product, and why is that a business decision as much as a legal one?*
8. *A container cannot reach the instance metadata service although the host can. Where do you look first, and why does a hop limit exist at all?*
9. *Your verification script found a production bug after the code had passed CI and been merged. What does that say about what CI can and cannot prove?*

Step 8 – /learn Documentation Site + PDF Export (2026-10-19)

Verified by: `node scripts/build-learn.js` run twice in a row (identical output, no drift) · one page per source document plus a landing page and the print guide (13 documents + README → 16 pages) · `./scripts/build-pdf.sh` rendering `learn/gamehub-guide.pdf` (68 A4 pages) through headless Chrome · a spot-check that the GitHub-style tables in the step notes render as real tables, not pipe soup · the CI drift check detecting a stale site when a one-line README edit is not regenerated, and passing on the committed `learn/`.

What was built

The step notes in `docs/` and the `README.md` used to live only as Markdown in the repository. They are now a small static site under `learn/`, plus a single printable guide, generated by two dependency-free scripts.

Piece	Implementation	Why not the alternative
Site generator [<code>scripts/build-learn.js</code>]	Node 22 CommonJS, built-ins only [<code>node:fs</code> , <code>node:path</code>] — reads <code>README.md</code> + <code>docs/*.md</code> , writes <code>learn/index.html</code> , one <code>learn/<slug>.html</code> per document, <code>learn/guide.html</code> , <code>learn/style.css</code> and <code>learn/fonts/</code>	<i>A static-site framework</i> (Eleventy, Jekyll, VitePress) → hundreds of transitive packages and a build toolchain to keep updated for a handful of Markdown files; <i>rendering Markdown in the browser</i> → every visitor downloads a parser and the page is empty without JavaScript
Hand-written Markdown renderer [~200 lines]	Headings, paragraphs, bold/italic/inline code, fenced code, links, autolinks, images, ordered/unordered lists, nested lists, blockquotes, task-list checkboxes, horizontal rules and GitHub-style tables	<i>marked / markdown-it</i> → a runtime dependency (and a supply-chain surface) for rules this project's docs actually use; the docs lean on tables, so table support is the part that had to be right
Escaping	Every block escapes text [<code>&</code> , <code><</code> , <code>></code> , <code>"</code>] before inline formatting; code spans are extracted first and re-escaped on restore	<i>Trusting the source</i> → <code>README.md</code> legitimately contains <code><https://...></code> links and <code><canvas></code> inside code spans; the renderer has to show them literally without ever emitting raw injected HTML
PDF export [<code>scripts/build-pdf.sh</code>]	Runs the generator, then prints <code>learn/guide.html</code> with headless Chrome/Chromium [<code>--print-to-pdf</code>] into <code>learn/gamehub-guide.pdf</code>	<i>A PDF library</i> (PDFKit, Puppeteer) → Puppeteer bundles a browser; PDFKit lay out Markdown by hand. The machine already has a browser, so the browser does the page layout
Design [<code>learn/style.css</code>]	The app's own arcade language — dark panels, cyan/magenta accents, Chakra Petch + Press Start 2P — with a sidebar listing every document and a PDF download button	<i>GitHub-flavoured CSS</i> → the docs are part of this project and should look like it

The generated `learn/` directory is committed, not ignored: it is a build output, but a small, reviewable one, and it means the site can be published without running a generator in the deploy path.

How the guide becomes a PDF

`learn/guide.html` is a second rendering of the same documents, print-optimised rather than screen-optimised: an A4 `@page` rule, a cover page with the project title, a contents list, and each document starting on a fresh page with its `<h1>` as the section break. `scripts/build-pdf.sh` locates the browser [`/Applications/Google Chrome.app/...` on macOS, then `chromium` / `google-chrome` on Linux, overridable with `CHROME_BIN`] and prints that one file. Because the guide reuses the same renderer as the site, the PDF cannot drift from what the site shows.

Things that had to be got right

1. **Tables are the whole point.** The step notes and the README are mostly `| ... | ... |` tables — what was built, why, and why not the alternative. A renderer without GFM table support would have produced an unreadable site, so the table parser (header, separator, alignment from `:---`) was written and checked first.
2. **Inline code inside emphasis.** `** GET /api/games/:id **` must produce a `<strong>` wrapping a `<code>`, not literal asterisks. Code spans are pulled out into placeholders before escaping and put back at the end, so they can never be re-parsed as emphasis.
3. **Fonts stay self-hosted.** The site and the PDF both reference `fonts/press-start-2p-latin-400.woff2` and the Chakra Petch faces relatively, and the generator copies them out of `frontend/fonts/` into `learn/fonts/`. There is no Google Fonts import anywhere — the same GDPR reasoning as Phase 2.
4. **A second run changes nothing.** The generator is deterministic: the same Markdown produces the same bytes, so re-running it never shows up as a spurious diff and CI could gate on it if the docs were generated there.
5. **The success status was the bug.** `/learn/` answered **200 with the game hub's landing page**, and the deploy's smoke test passed on it because it asserted a string [`GameHub`] that both pages contain. The check is now specific: it looks for `readme.html`, which is on the docs landing page and nowhere else. A verification step is only as strong as its most specific assertion.

Concepts to understand cold

- **A hand-written renderer vs a framework:** what you actually own when you write ~200 lines of Markdown parsing, and when that trade stops being worth it
- **Escaping order:** why you extract code spans, escape, then format in a single left-to-right pass (so generated HTML is never re-scanned)
- **Print CSS:** `@page size: A4`, `break-before: page`, and `break-inside: avoid` on table rows — how a web page becomes a document that paginates
- **Static hosting with relative paths:** why every asset is referenced relatively [`style.css`, `fonts/...`, `gamehub-guide.pdf`] so the same files work at `/` locally and under `/learn/` on CloudFront

How `/learn/` is served

The site is plain static files with relative references, so it is published under the `learn/` prefix of the *same* S3 bucket that serves the app, and the *same* CloudFront distribution serves it at `<distribution>/learn/` — no second bucket, no second CDN, no new origin. CloudFront already knows how to serve objects the bucket does not make public (private bucket + Origin Access Control), so the documentation inherits that.

`scripts/build-frontend.sh` copies `learn/` into `build/frontend/learn/`, so the deploy's root sync (which uses `--delete`) publishes the docs and keeps them: the prefix is part of the artifact being synced on every

build, which is why no separate prefix-scoped sync is needed and why a delete-sync of the root cannot remove the docs by accident.

Two guards keep the published site truthful:

- **A drift check in CI:** the `manifests` job re-runs the generator and fails on any diff against the committed `learn/`, and asserts that every document in `README.md` and `docs/*.md` has a page. A generated artifact that is committed and never checked is a generated artifact that goes stale – and this one already had, when `README.de.md` and the step notes were committed without regenerating it.
- **The PDF is excluded from that comparison**, because it is rendered by a headless browser and is not byte-reproducible on a runner. `make learn` regenerates the site and the PDF together, and the deploy publishes whatever is committed.

A directory URL is not a file

`/learn/` first answered **200 with the game hub's landing page**. An S3 origin reached through an Origin Access Control is a REST origin: it has no notion of an index document, so a request for the key `learn/` is a 403 – and the distribution's custom error responses turn a 403 into `/index.html` with a 200. A wrong page wearing a success status is worse than an error, because nothing reports it; the deploy's own smoke test passed on it.

The fix is a twelve-line CloudFront Function on the viewer request of the static cache behaviour: a URI ending in `/` gets `index.html` appended, and an extensionless last segment is treated as a directory so `/learn` works as well as `/learn/`. It returns `/api/*` untouched, so its correctness does not depend on cache-behaviour routing. Twelve URI cases are exercised locally before the function is deployed, and the reasoning – including why a real `learn/` object in the bucket was rejected – is ADR-017 in `decisions.md`.

Interview questions this unlocks

1. "You generated documentation from Markdown – why not use a static-site framework?" (dependency budget vs the rules this project actually needs)
2. "How do you stop user- or author-supplied Markdown from injecting HTML?" (escape order and code-span placeholders)
3. "How did you turn a web page into a paginated A4 PDF?" (print CSS + a headless browser that already exists on the machine)
4. "Two independent websites live in one S3 bucket – how do you deploy one without deleting the other?" (prefix-scoped sync; never `--delete` the root)
5. "Your health check passes and the page is wrong. How is that possible?" (a custom error response turning an origin 403 into a 200, and a smoke test whose only assertion held for both pages)

Architecture decisions (ADRs)

Every non-obvious choice in this project, in the same shape: **context**, **decision**, **consequences** (including what it costs), **rejected alternatives**. The step documents [docs/step-*.md] tell the story of each phase; this file is the index of the decisions that story is made of.

#	Decision	Status	Phase
001	One Compose file for local development and production	Accepted	0
002	MySQL in a container, not RDS	Accepted	0
003	CloudFormation, not Terraform	Accepted	4
004	Keyless deploys with GitHub OIDC	Accepted	4
005	Self-hosted fonts, zero third-party requests	Accepted	2
006	Metrics derived from structured nginx logs	Accepted	5
007	The synthetic check is a scheduled Lambda writing a log line	Accepted	5
008	kind with the Compose images side-loaded	Accepted	6
009	Argo CD with <code>automated</code> + <code>prune</code> + <code>selfHeal</code>	Accepted	6
010	JWT verification with <code>node:crypto</code> and the pool's JWKS	Accepted	4
011	One personal-best row per <code>(sub, game)</code> , not a score history	Accepted	7
012	SigV4 by hand over IMDSv2 credentials, no AWS SDK	Accepted	7
013	A checksummed migration runner, not only <code>init.sql</code>	Accepted	7
014	Deploy on a successful CI run, not on the push	Accepted	7
015	No package manager in the runtime image	Accepted	7

ADR-001

One Compose file for local development and production

Context. The project has to run on a laptop with one command and on an EC2 host. Two definitions of "the stack" drift.

Decision. `docker-compose.yml` is the only definition. The host clones the repository and runs the same file, the same images and the same environment variable names.

Consequences. A bug found locally is a bug in production, and the EC2 host needs Git + Docker and nothing else. The host runs MySQL in a container, so the database is stateful inside Docker – acceptable at this size, wrong at scale.

Rejected. A separate production compose file (two files to keep in sync); ECS/Fargate (more moving parts than the app).

ADR-002

MySQL in a container, not RDS

Context. A managed database with backups and patching is what a real service would use; it is also the largest line item in a small AWS bill.

Decision. MySQL 8.4 in a container on the host, seeded from `db/init.sql` on an empty volume.

Consequences. Zero cost, and the schema is reproducible from the repository. No automated backups, no multi-AZ, no managed patching — and the project says so rather than implying otherwise. The migration path to RDS is a phase of its own.

Rejected. RDS (cost, and it would have hidden the one-command local story); SQLite (removes the realistic connection-pool and migration surface).

ADR-003

CloudFormation, not Terraform

Context. Infrastructure as code for five stacks: Cognito, S3+CloudFront, the EC2 host, observability and the OIDC provider. (A sixth, `gamehub-dns`, was added later for the custom domain's certificate — it lives in us-east-1 because that is the only region CloudFront accepts certificates from; see ADR-018.)

Decision. Plain YAML CloudFormation templates in `infra/`, one per stack, wired with outputs.

Consequences. No state file to store, lock or lose, and no third-party provider to trust. The templates are verbose, cross-stack references are awkward, and `!If` / `!Sub` are not a programming language.

Rejected. Terraform/OpenTofu (an extra state backend to run and secure for five small stacks); CDK (a build step and a runtime for what is mostly declarative configuration).

ADR-004

Keyless deploys with GitHub OIDC

Context. CI has to reach S3, CloudFront, EC2 (via SSM) and ECR-less hosts.

Decision. A GitHub OIDC provider plus an IAM role restricted to this repository, whose trust policy pins the repo and the deploy branch. Credentials are short-lived tokens from `sts:AssumeRoleWithWebIdentity`.

Consequences. Nothing to rotate, nothing to leak, and the role is the audit trail. The trust policy must be kept in step with repository renames — a rename breaks deploys until the condition string is updated.

Rejected. Long-lived IAM user access keys in GitHub secrets (a rotation problem and a leak risk); a self-hosted runner (cost, and a runner is not a credential strategy).

ADR-005

Self-hosted fonts, zero third-party requests

Context. The UI needs a game-style typeface. The obvious way is `fonts.googleapis.com`.

Decision. The four OFL-licensed woff2 files are committed under `frontend/fonts/` and served from the same origin; the `/learn` site does the same.

Consequences. No visitor IP is disclosed to a third party at page load, which German courts have held needs consent, and no CDN dependency. The woff2 files (~120 KB total) live in the repository, and each font's licence is committed next to it.

Rejected. Google Fonts (a data-protection problem for a German audience); a system-font stack (loses the arcade look the project is about).

ADR-006

Metrics derived from structured nginx logs

Context. The SLOs need request rate, 5xx rate and latency percentiles. A plaintext access log can only be grepped.

Decision. nginx writes JSON (`frontend/nginx-log-format.conf`), the logs go to CloudWatch through the agent, and *metric filters* extract the numbers. The `log_format` lives in its own file because it is an http-level directive and the repository's nginx config is a `server` block.

Consequences. The metric definition is a CloudFormation pattern next to the alarm that uses it, and the log line is the source of truth. Metric filters are eventually consistent (~1 minute), a non-matching event still emits the filter's default value (so `Sum` is safe and `Average` is diluted), and percentiles can only be approximated from bucketed filters.

Rejected. An agent that parses logs into custom metrics (a second parser to maintain); application-level metrics only (loses nginx's view of status codes and timing).

ADR-007

The synthetic check is a scheduled Lambda writing a log line

Context. Availability should be judged from outside, on a schedule, and it should be visible on a dashboard.

Decision. EventBridge runs a small Lambda every 5 minutes; the Lambda requests the real site, asserts the result, and writes **one** JSON log line. A metric filter turns pass/fail into a metric, and the alarm is `Sum < 1` with `TreatMissingData: breaching` – silence is a failure, not an unknown.

Consequences. No per-check charge and a single code path from the check to the alarm. The threshold semantics are unusual and are written down (`docs/step-5-observability.md`) precisely because `Sum < 1` looks wrong at a glance.

Rejected. CloudWatch Synthetics canaries (a per-run cost and a second deployment model); external uptime services (a third party, and no path into the dashboard).

ADR-008

kind with the Compose images side-loaded

Context. Kubernetes manifests need to be exercised somewhere real.

Decision. A single-node kind cluster [`k8s/kind-config.yaml`], loaded with the **same** images the Compose stack builds [`imagePullPolicy: Never`], with the NodePort published on host 8081 so Compose keeps 8080.

Consequences. One image, two orchestrators — a manifest change is tested against the artifact that also runs locally. kind needs a capable container VM (4 CPU / 6 GiB Colima here), and a `Never` pull policy fails fast when the image was not side-loaded.

Rejected. minikube (a VM plus a driver argument per laptop); EKS (real money for a demo); a separate Kubernetes image build (two artifacts that can drift).

ADR-009

Argo CD with `automated` + `prune` + `selfHeal`

Context. "GitOps" is a claim about who owns cluster state.

Decision. An Argo CD `Application` points at `k8s/` on `main` with `syncPolicy.automated { prune: true, selfHeal: true }`.

Consequences. The merge is the deploy, and a manual `kubectl` change is reverted by the controller — verified by scaling a Deployment by hand and watching it return to the declared replica count within 90 seconds. A mistake in `k8s/` is now a cluster-wide change on merge, which is exactly why CI validates the manifests (kubecnf) and the generated seed ConfigMap before the merge happens.

Rejected. `kubectl apply` from a pipeline (state is whatever the last successful run left behind); Flux (equivalent here, and Argo CD's UI makes the demo legible).

ADR-010

JWT verification with `node:crypto` and the pool's JWKS

Context. The API must verify Cognito ID tokens: RS256 signature, `iss`, `aud`, `exp`, `token_use`, and a key id that rotates.

Decision. Fetch the JWKS, cache it, and verify the RS256 signature with `node:crypto`'s `createPublicKey` / `verify` — no `jsonwebtoken`, no AWS SDK.

Consequences. The backend has no authentication dependency to audit, and the verification rules are visible in one file. The JWKS cache, key rotation and the `kid`-miss refetch path are code this project owns and tests (a forged token, an expired token, a wrong-pool token and a non-ID token each get a `401`).

Rejected. `jsonwebtoken` (fine, but it hides the checks that are worth demonstrating); API Gateway's Cognito authorizer (removes the app's own authorization surface entirely).

ADR-011

One personal-best row per `(sub, game)`, not a score history

Context. "Remember my high score" could be a row per play.

Decision. `personal_bests` has the primary key `(user_sub, game)` and updates with `GREATEST(score, VALUES(score))`. The player is referenced by the Cognito `sub`; the table holds no email address.

Consequences. The feature works with one row per player per game, so the data a breach would expose is smaller than it needed to be, and deletion `[ DELETE /api/me ]` is a single indexed delete. The cost is real: there is no history, so no leaderboard-over-time, no graphs, no "your last 10 games" — features that would require collecting personal data the app currently has no reason to hold.

Rejected. An append-only `scores` table (a behavioural record collected for a feature that does not exist); keying on email (copying an identifier into a second system for no benefit).

ADR-012

SigV4 by hand over IMDSv2 credentials, no AWS SDK

Context. Account deletion needs `cognito-idp:AdminDeleteUser`, which is an admin API: the player's JWT cannot authorize it, so the backend must sign its own AWS request.

Decision. `backend/cognito.js` fetches temporary credentials from the instance metadata service with IMDSv2 (a `PUT` for a token, the token on every `GET`) and signs the request with a hand-rolled SigV4 implementation.

Consequences. No SDK in the image, no long-lived keys on the host, and the signing algorithm is asserted in tests (canonical request, signed-header list, payload binding, session token included in the signature). The maintenance cost is real: the code is this project's to keep correct, and the tests are the reason that is acceptable.

Rejected. `@aws-sdk/client-cognito-identity-provider` (tens of megabytes and a credential-provider chain for one call); IMDSv1 (any SSRF becomes role theft); a long-lived IAM user key (a secret on disk to rotate and leak).

ADR-013

A checksummed migration runner, not only `init.sql`

Context. `db/init.sql` runs once, on an empty data volume. A live database that already has data can never be changed by editing it.

Decision. `scripts/migrate.sh` applies files from `db/migrations/` in order, recording each in a `schema_migrations` ledger **with a checksum**, and refuses a migration whose file changed after it was applied. It reaches MySQL through `docker compose exec` on a host and the `mysql` client in CI.

Consequences. The same script is used by the deploy, by CI, and by a developer, and the ledger answers "what has been applied here?". The rule that a migration is immutable once applied is a discipline the tool only enforces for files it has seen before.

Rejected. Relying on `init.sql` plus manual `ALTER` statements on the host (undocumented, unrepeatable); a framework (a dependency for ~100 lines, and the runner is a teaching artifact here).

ADR-014

Deploy on a successful CI run, not on the push

Context. A push to `main` should reach production, but only after the tests have judged it – and, with concurrent merges, not a later commit than the one tested.

Decision. `.github/workflows/deploy.yml` triggers on `workflow_run` for the CI workflow completing on `main` with `conclusion == 'success'`, and checks out `github.event.workflow_run.head_sha`.

Consequences. Production receives exactly the commit CI passed. Path filters are gone, so a documentation-only merge re-deploys identical content (idempotent, and worth the safety). The pipeline ends with a smoke test against the public CloudFront URL.

Rejected. Deploying on `push` [a race against CI]; a single workflow with jobs depending on each other [couples a 20-minute test run to a deploy trigger].

ADR-015

No package manager in the runtime image

Context. A Trivy scan of the backend image reported 10 HIGH findings, all inside `/usr/local/lib/node_modules/npm` – the package manager's own bundled dependency tree, in an image whose entrypoint is `node server.js`.

Decision. Remove npm, npx and corepack from the runtime layer after `npm ci` has installed the production dependencies.

Consequences. A smaller image, fewer findings, and no package manager for an attacker to use if the process is ever compromised. The image can no longer be used for interactive dependency debugging [`docker run ... npm ...` needs a build stage or a different image], which is the intended trade.

Rejected. Accepting the findings with a `.trivyignore` [the vulnerable code is not needed at runtime, so removing it beats excusing it]; moving to a distroless base [attractive, but it removes the shell this project uses for `HEALTHCHECK` and for diagnosing the container].

ADR-016

A post-deploy check must assert what production actually does

Context. The first version of the deploy's `verify` job fetched `/api/games` without a token and asserted 200 with 12 games. Against production it got 401 – the auth gate doing exactly its job – and failed the deploy. Nothing was wrong with the deployment: the check encoded an assumption that stopped being true when the catalogue moved behind the sign-in gate, and nothing re-examined the check when that happened. The identical wrong assumption appeared a second time, independently, in the load tool's success metric.

Decision. Anonymous requests are asserted to be *refused* [401 on every protected route], and anything needing real data signs in as a dedicated smoke identity – `smoke-test@gamehub.example`, credentials in the repository secrets `SMOKE_USER_EMAIL` / `SMOKE_USER_PASSWORD` – and then uses that token exactly as a browser would. The check also covers `/learn/` and the PDF.

Consequences. The green tick means "a player can do what a player does": edge, JWKS verification, database and docs, not merely "something answered". It depends on one Cognito identity and two repository secrets; if they are missing the job fails loudly, which is the right failure mode for a gate. A test that cannot fail for the right reason is not a test.

Rejected. Making `/api/games` public again so the old assertion would pass (letting the check reshape the product); asserting only that the routes answer (a 401 and a 200 are both "answering" — the cheapest way to have a green check that proves nothing).

ADR-017

Directory URLs are rewritten at the edge, not stored as folder objects

Context. `/learn/` answered **200 with the game hub's landing page**. An S3 origin behind an Origin Access Control is a REST origin and has no index document: a request for the `learn/` key is a 403, and the distribution's custom error responses turn a 403 into `/index.html` with a 200. The deploy's smoke test passed on that wrong page, because its only assertion [`GameHub`] is true of both pages.

Something has to resolve `/learn/`. Three candidates: write `/learn/index.html` in every link, store a real object under the key `learn/`, or rewrite the URI at the edge.

Decision. A CloudFront Function on the viewer request of the static cache behaviour appends `index.html` to a URI that ends in `/`, and treats an extensionless final segment as a directory so `/learn` works too. It returns `/api/*` unchanged, so the function is correct independently of which cache behaviour it is attached to. Its ten-to-twelve URI cases are exercised locally before it is published.

Consequences. `/learn/`, `/learn` and every deep link work; the bucket stays dumb; the cost is nil at this size (the CloudFront Functions free tier is two million requests a month against a few thousand here). The trade is one more piece of logic between the user and the object, so it fails open — an unmatched URI behaves exactly as it did before — and it is small enough to read in one sitting.

Rejected. Uploading the landing page *also* as the key `learn/` (S3 permits it, but `aws s3 sync --delete` does not know it is meant to exist, so it would be deleted and restored on every deploy — a one-second outage per release in exchange for saving twelve lines); linking to `/learn/index.html` everywhere (works, and the docs' own links do use explicit `.html`, but `/learn/` is the URL a person types or bookmarks, and a landing page that renders the wrong site is a bug however it was linked).

ADR-018

The site answers on a custom domain — and the CloudFront name keeps working

Context. The site was reachable only at `d2qf9zxehny6bk.cloudfront.net`. The zone `worldofvpc.online` already existed with another project on the apex, so `gamehub.worldofvpc.online` was the obvious subdomain. The expected cost was breaking the CloudFront name: the distribution redirects HTTP to HTTPS, so if a custom certificate replaced the default one, the old address would fail at the TLS handshake and any link to it would simply break.

Decision. Serve `gamehub.worldofvpc.online`: an ACM certificate issued in us-east-1 by a small dedicated stack [`infra/dns.yaml`] and validated over DNS, an alias on the distribution, and IPv4/IPv6 alias records pointing at it. Prefer the custom name wherever a URL is written down — README, docs, the load-test commands and the deploy's smoke test — and treat the CloudFront name as an implementation detail.

Consequences. The expectation was wrong in a pleasant way, and it was measured rather than assumed: CloudFront chooses the certificate by SNI, so the alias gets the certificate for `gamehub.worldofvpc.online` while the distribution's own name keeps getting `*.cloudfront.net`. **Both names answer 200**, so no existing link breaks and the historical verification records stay exactly as they were written. The real cost is a third stack (in us-east-1, because CloudFront only accepts certificates from there), a hosted-zone id in the site stack, and the standing obligation to rename *everywhere at once* — a half-renamed project is worse than an unnamed one. Certificates are free; alias records are not charged for.

Rejected. A second distribution for the custom name (a duplicate of the whole configuration — cache behaviours, the directory-index function, the origin access control — to avoid one alias); email-validated certificates (a human has to click a link, and the failure mode is an expired certificate in production); and rewriting the historical records that name the CloudFront domain, which would have falsified evidence that was accurate when it was captured.

Load test — what a single t3.micro actually does (2026-10-19)

Verified by: two runs of `scripts/load-test.js` against the live deployment on 2026-10-19 at 09:35–09:36 UTC (4,000 requests total), with the **origin-side** numbers read back from the app's own request log [`/gamehub/prod/app`] and nginx's access log [`/gamehub/prod/nginx-access`] in CloudWatch, and the instance's CPU, memory and all eight alarm states read back from CloudWatch for the same window.

Why this exists, and what it is not

A load test is only useful if it is honest about what it measured. This one was built to answer a narrow question — *does a two-vCPU burstable instance with a containerised MySQL comfortably serve this application?* — and it deliberately does **not** try to find the breaking point.

Three things make the result trustworthy rather than flattering:

1. **Three vantage points, not one.** The client measures end-to-end latency (which, from a home connection, is mostly *the network*). nginx's `request_time` measures what the origin spent. The backend logs its own `duration_ms` per request, which is the handler alone. Reporting only the first number would have said "180 ms per request" and meant almost nothing.
2. **The mix is weighted and declared.** 50 % static site [`/` , served by CloudFront], 30 % API list [`/api/games` , a database read], 20 % liveness [`/api/health` , no database]. The weights are in the script.
3. **Each endpoint declares its expected status.** 401 from the gated catalogue route is the auth gate working, not an error — so the anonymous run scores 401 as *expected*, and a 200 there would be the finding. Without that, the first run "failed" with a 70 % success rate while the system was perfect.

The client's view

Run A is anonymous (the catalogue answers 401); Run B carries a real ID token, so `/api/games` reaches MySQL. Both: 2,000 requests, concurrency 20, keep-alive, one client.

Run A — anonymous (the edge and the gate)

- **Target:** `https://d2qf9zxehny6bk.cloudfront.net` [the distribution's own name; the site's canonical name is `gamehub.worldofvpc.online` , and both are served — see ADR-018]
- **Duration:** 19.17 s · **Throughput:** 104.32 req/s
- **Responses as expected:** 100 % · **Unexpected:** 0 · **Connection errors:** 0

Latency	ms
min	161.19
mean	190.78
p50	180.25
p90	193.81
p95	234.46
p99	348.87
max	1103.22

Endpoint	Requests	p50 (ms)	p95 (ms)	Expected	Unexpected
/	1000	186.08	242.57	200	0
/api/games	600	174.52	193.52	401	0
/api/health	400	177.31	233.69	200	0

Status distribution: {"200": 1400, "401": 600}.

Run B – authenticated (the database path is exercised)

- **Duration:** 19.19 s · **Throughput:** 104.23 req/s
- **Responses as expected:** 100 % · **Unexpected:** 0 · **Connection errors:** 0

Latency	ms
min	159.34
mean	190.73
p50	181.47
p90	195.34
p95	234.48
p99	265.04
max	932.88

Endpoint	Requests	p50 (ms)	p95 (ms)	Expected	Unexpected
/	1000	184.77	242.28	200	0
/api/games	600	178.04	230.18	200	0
/api/health	400	177.97	232.75	200	0

Status distribution: {"200": 2000}.

Read these numbers carefully: they are dominated by one client's network. The distance from the test machine to the nearest CloudFront edge is ~175 ms, so ~180 ms of every request is travel time, not work.

Comparing the two runs shows this plainly: adding a **real database read** to 600 requests changed p50 by 1.2 ms and p95 not at all. The origin's own numbers below are the meaningful ones.

The origin's view

nginx logs `request_time` in **seconds** (the tables below are converted to milliseconds), and the backend logs `duration_ms` per request.

Run A – anonymous

What	n	avg (ms)	p50 (ms)	p95 (ms)	p99 (ms)	max (ms)
nginx: <code>/api/games</code> → 401	599	1.00	1.00	2.00	7.00	14.00
nginx: <code>/api/health</code> → 200	400	1.00	1.00	2.00	3.00	12.00
backend: <code>/api/health</code> handler	401	0.456	0.400	0.700	1.200	2.600
nginx: <code>/</code> (static)	0	–	–	–	–	–

Run B – authenticated

What	n	avg (ms)	p50 (ms)	p95 (ms)	p99 (ms)	max (ms)
nginx: <code>/api/games</code> (proxied, DB read)	600	2.00	2.00	4.00	10.00	28.00
nginx: <code>/api/health</code>	400	1.00	1.00	2.00	6.00	13.00
backend: <code>/api/games</code> handler (DB read)	600	1.242	1.000	1.900	4.000	22.300
backend: <code>/api/health</code> handler	403	0.438	0.400	0.500	1.500	6.800
nginx: <code>/</code> (static)	0	–	–	–	–	–

Three things stand out:

- **The static half never touched the origin.** 2,000 of the 2,000 `/` requests across both runs were served from the CloudFront cache – no nginx entry exists for them. Half of the traffic mix cost the instance nothing.
- **A database read costs about a millisecond.** `SELECT id, name, category, description FROM games ORDER BY id` over 12 rows: p50 1.0 ms, p99 4.0 ms, max 22.3 ms across 600 requests. The proxy adds roughly another millisecond.
- **The client saw ~180 ms while the origin spent ~2 ms.** Roughly 99 % of the measured latency is the path between the test machine and the edge.

The instance while it was being hit

Metric	Baseline	During the runs
EC2 <code>CPUUtilization</code> (1-min max)	~2.7 %	4.55 %, peak 6.97 %
<code>mem_used_percent</code>	~41.6 %	42.0 – 43.2 %
Instances with a state change	–	none

CPU peaked at **7 %** while the origin absorbed ~52 requests per second (1,000 origin-bound requests per run over 19 s) with DB-backed p95 at 2 ms. The test does not find this instance's limit; it does not come close to it.

The write path was measured separately, during the Phase 7 verification, and is the only place where the app talks to AWS:

Request	n	avg (ms)	max (ms)	What dominates
POST /api/scores	2	79.65	149.30	MySQL upsert + round trip
DELETE /api/me	1	239.30	239.30	the SigV4 <code>AdminDeleteUser</code> call to Cognito

That is the honest shape of the system: reads are ~1 ms, and anything that crosses into a control-plane API costs hundreds of milliseconds.

What the alarms did

Nothing — and that is the correct result. All eight alarms stayed `OK`, and CloudWatch reports **no alarm state change in the 45 minutes around the runs**:

Alarm	State during/after the runs
gamehub-prod-synthetic-check-failed	OK
gamehub-prod-http-5xx	OK
gamehub-prod-api-latency-p99	OK
gamehub-prod-instance-status-check-failed	OK
gamehub-prod-cpu-high	OK
gamehub-prod-cpu-credit-balance-low	OK
gamehub-prod-memory-high	OK
gamehub-prod-disk-high	OK

Two successor questions follow from this, and both are answered elsewhere: *would the alarms fire if the instance were actually struggling?* — the alarm path was exercised and the thresholds verified separately in [docs/step-5-observability.md](#); and *is the latency alarm measuring the right thing?* — it watches the origin's `request_time` metric, not the client's end-to-end time, which is why a slow client link (this test) does not page anyone.

What this does NOT prove

- **One client, one location.** Concurrency 20 from a single machine is not distributed load; CDN edge performance for a user in another country is untested.
- **No saturation point.** Neither run pushed CPU past 7 %, so the numbers say "comfortable at this level", not "this is the ceiling".
- **A burstable instance is burstable.** The `t3.micro` earns CPU credits while idle and spends them under load. A 19-second burst is exactly what the family is designed for; a *sustained* load test is a different experiment, and the `gamehub-prod-cpu-credit-balance-low` alarm exists because of it.
- **A warm static cache.** The static half of the mix was a cache hit throughout. A cold-cache storm is a CloudFront capacity question, not an instance one, and is not covered here.
- **The write path is sampled, not load-tested.** Two `POST /api/scores` and one `DELETE /api/me` are enough to characterise one request, not to make a claim about write throughput.

- **No adversarial traffic.** No rate limiting, no abuse pattern, no large payloads — those belong to a security review, not a capacity one.

Cost

4,000 requests from CloudFront plus ~2,000 origin-bound requests and a few hundred kilobytes of transfer. CloudFront's free tier is 1 TB and 10 million requests per month; the instance is billed whether it is busy or idle. The marginal cost of this test was **effectively zero**, and nothing here approaches a paid tier.

Reproduce it

```
make load-test SITE_URL=https://gamehub.worldofvpc.online REQUESTS=2000 CONCURRENCY=20

# authenticated (the API then reaches MySQL); get an ID token first, e.g. with
# scripts/verify-gdpr.sh's sign-in steps, then:
node scripts/load-test.js --url "$SITE_URL" --requests 2000 --concurrency 20 --token
"$ID_TOKEN"

# the origin's side of the same window:
aws logs filter-log-events --log-group-name /gamehub/prod/app \
  --filter-pattern '"duration_ms"' --start-time <epoch-ms> --end-time <epoch-ms>
aws logs filter-log-events --log-group-name /gamehub/prod/nginx-access \
  --start-time <epoch-ms> --end-time <epoch-ms>
```

Runbook — GameHub (prod)

One section per alarm. Each one says what the alarm means, how to confirm the cause with a command, and what to do about it. Everything here is done over **SSM** — there is no SSH access to the host by design.

0. Ground rules

```
export AWS_REGION=eu-central-1
export INSTANCE_ID=$(aws ec2 describe-instances --region "$AWS_REGION" \
  --filters Name=tag:Name,Values=gamehub-prod* \
  --query "Reservations[].Instances[].InstanceId" --output text)
# The site's canonical name. The distribution's own domain name also still
# answers, with CloudFront's default certificate — see ADR-018.
export SITE_URL=https://gamehub.worldofvpc.online
```

Run a command on the host (no keys, no SSH — IAM decides who may do this):

```
aws ssm send-command --region "$AWS_REGION" --instance-ids "$INSTANCE_ID" \
  --document-name AWS-RunShellScript \
  --parameters 'commands=["docker compose -f /opt/gamehub/docker-compose.yml ps"]' \
  --query Command.CommandId --output text
```

Check every alarm at once:

```
aws cloudwatch describe-alarms --region "$AWS_REGION" \
  --alarm-name-prefix gamehub-prod \
  --query "MetricAlarms[][AlarmName,StateValue]" --output table
```

The general shape of the response is always the same: **state** → **evidence** → **action**, and only ever through code, never the console.

1. gamehub-prod-synthetic-check-failed

Means: the external check could not get a `200` from the site or from `/api/health` within 10 seconds. This is the closest thing to "users cannot use the site".

Confirm:

```
curl -sS -o /dev/null -w '%{http_code} %{time_total}s\n' $SITE_URL/
curl -sS -o /dev/null -w '%{http_code} %{time_total}s\n' $SITE_URL/api/health
aws lambda invoke --region "$AWS_REGION" \
  --function-name gamehub-prod-canary --log-type Tail /dev/stdout | tail -5
```

Act:

Evidence	Likely cause	Action
<code>/</code> fails, <code>/api/health</code> fine	CloudFront/S3 path	check the distribution status, then re-run the deploy workflow [<code>workflow_dispatch</code>]
both fail	the EC2 host is down	go to section 4 (instance status)
<code>/api/health</code> fails, <code>/</code> fine	the Express container is unhealthy	section 3, then <code>docker compose up -d</code> on the host

2. gamehub-prod-http-5xx

Means: more than 5 responses with `status >= 500` in 5 minutes behind nginx.

Confirm – the log line is JSON, so this is a query rather than a grep:

```
# CloudWatch Logs Insights, /gamehub/prod/nginx-access
fields ts, uri, status, request_time, remote_addr
| filter status >= 500
| sort ts desc
| limit 20
```

Then, on the host:

```
docker compose -f /opt/gamehub/docker-compose.yml logs --tail=200 backend
```

Act: a 5xx caused by the last deploy → `git revert` on `main` and let the pipeline republish. A 5xx from the database [`ER_...` in the log] → check `docker compose ps db` and the MySQL healthcheck; the app returns 503, not a crash, so the frontend stays up.

3. gamehub-prod-api-latency-p99

Means: p99 of `/api/*` above 1.5 s for 10 minutes.

Confirm:

```
# CloudWatch Logs Insights
fields ts, uri, request_time, upstream_time
| filter uri like /^\/api/
| stats pct(request_time, 99) as p99, avg(request_time) as avg, count(*) as n by uri
| sort p99 desc
```

The gap between `request_time` and `upstream_time` tells you who is slow: nginx/TLS-side or Express.

Act:

Cause	Action
One expensive endpoint	fix the query or add an index – the catalogue read and the scores read are the only DB paths
CPU saturation (see <code>cpu-high</code>)	the instance is the bottleneck: stop a concurrent <code>docker compose build</code> and confirm the credit balance is recovering
MySQL swapping	memory alarm follows; the host has 2 GB swap and 1 GB RAM, so a heavy query can push MySQL into swap

4. `gamehub-prod-instance-status-check-failed`

Means: the hypervisor cannot reach the instance, or the OS is not responding. Everything else is noise until this is clear.

Confirm:

```
aws ec2 describe-instance-status --region "$AWS_REGION" \
  --instance-ids "$INSTANCE_ID" --include-all-instances \
  --query "InstanceStatuses[].[InstanceStatus.Status,SystemStatus.Status]"
aws ec2 get-console-output --region "$AWS_REGION" \
  --instance-id "$INSTANCE_ID" --output text | tail -30
```

Act: a *system* status failure is AWS-side – stop and start the instance (the elastic IP keeps the address stable) and let the user-data boot the stack again. An *instance* status failure with an OOM message in the console output means the stack ran out of memory: the container restart policies bring it back, but the fix is a real one (see `cpu-credit-balance-low`).

5. `gamehub-prod-cpu-high` / `cpu-credit-balance-low`

Means: the burstable instance is out of headroom. `CPUUtilization` above 85 % for 15 minutes, or fewer than 20 CPU credits left.

Confirm:

```
aws cloudwatch get-metric-statistics --region "$AWS_REGION" \
  --namespace AWS/EC2 --metric-name CPUCreditBalance \
  --dimensions Name=InstanceId,Value="$INSTANCE_ID" \
  --start-time "$(date -u -v-1H +%FT%TZ)" --end-time "$(date -u +%FT%TZ)" \
  --period 300 --statistics Average

# who is eating the CPU
aws ssm send-command --region "$AWS_REGION" --instance-ids "$INSTANCE_ID" \
  --document-name AWS-RunShellScript \
  --parameters 'commands=["docker stats --no-stream","uptime"]' \
  --query Command.CommandId --output text
```

Act: if a build is running, wait – a `docker compose build` on 2 vCPU is the expected cause. If it is sustained traffic, that is the signal to move the workload off the burstable class (see the roadmap), not to silence the alarm.

6. gamehub-prod-memory-high / gamehub-prod-disk-high

Means: memory above 90 % [a 1 GB instance running MySQL, Node and nginx] or the root filesystem above 85 % [Docker images and logs live there].

Confirm:

```
aws ssm send-command --region "$AWS_REGION" --instance-ids "$INSTANCE_ID" \
  --document-name AWS-RunShellScript \
  --parameters 'commands=["free -m","df -h /","du -sh /var/lib/docker /var/log/gamehub"]' \
  --query Command.CommandId --output text
```

Act:

Cause	Action
Memory	the swap file in the user-data is the buffer, not the fix: add a MySQL config file [mounted at <code>/etc/mysql/conf.d/</code>] with a smaller <code>innodb_buffer_pool_size</code> , or move the database off the box. The image's default is the right size for a real server and too greedy for a 1 GB host
Disk from Docker	<code>docker system prune -af --volumes</code> removes old layers [it will NOT touch the named volumes – verify with <code>docker volume ls</code> first]
Disk from logs	the agent ships logs with 7-day retention in CloudWatch; the local copies are rotated by Docker's <code>json-file</code> driver, so a runaway log is a code smell

7. Rollback

The pipeline is the only way changes reach the host, so the rollback is also a pipeline operation:

```
git revert <bad-sha>          # or: git revert --mainline 1 <merge-sha>
git push origin main
```

`deploy.yml` publishes the site and rolls the host to the reverted commit. The frontend image is built on the host from the checked-out commit, so "the previous version" is exactly the previous commit – no image registry needed. Database changes are the one thing a revert does not undo: the schema is additive [see `backend/schema.sql`], which is why it can be.

8. Escalation and evidence

- **Alarms:** `gamehub-prod-alarms` [SNS]. Subscribe with `aws sns subscribe --topic-arn <arn> --protocol email --notification-endpoint <you@example.com>` and confirm the mail.
- **Dashboard:** `gamehub-prod` [CloudWatch] – availability, error count, latency percentiles, host pressure, live log tables.
- **Logs:** `/gamehub/prod/nginx-access` [structured JSON, every request] and `/gamehub/prod/app` [container stdout/stderr, JSON from Express].
- **Audit:** CloudTrail records every `ssm:SendCommand`, so "who restarted the host at 03:00" has an answer.
- **Post-deploy check:** the `verify` job in `deploy.yml` asserts the *public* edge: the shell, `/learn/` and the PDF, the auth gate [401 without a token on every protected route], and – signing in as `smoke-test@gamehub.example` – a 12-entry catalogue and an identified user. Its credentials are the repository

secrets `SMOKE_USER_EMAIL` / `SMOKE_USER_PASSWORD` ; to rotate, run `aws cognito-idp admin-set-user-password --user-pool-id "$COGNITO_USER_POOL_ID" --username smoke-test@gamehub.example --password '<new>' --permanent` and update both secrets. If a rotated credential is not uploaded, the job fails — by design (see ADR-016).

- **Alarm notifications:** the topic has **one subscribed address** — the project owner's, passed to the observability stack as the `AlarmEmail` parameter, so CloudFormation owns the subscription and a manual `aws sns subscribe` would be drift. To change the recipient, re-deploy the stack with a different `AlarmEmail` and accept the confirmation mail: until it is accepted the subscription sits in `PendingConfirmation` and delivers nothing, which looks exactly like "the alarms are quiet tonight". Check it with `aws sns list-subscriptions-by-topic --topic-arn <AlarmTopicArn output>`. A subscription that was created and never confirmed is the failure mode to look for first when someone says alerts are not arriving.

Service level objectives – GameHub

These are the numbers the project holds itself to, how each one is measured, and what happens when one is missed. They are deliberately modest: this is a single t3.micro instance in one availability zone, and an SLO that claims more than the architecture can deliver is a lie in a markdown file.

1. Objectives

#	Objective	Target (rolling 30 days)	Measured by	Alarm
SLO-1	Availability – the public site answers <code>200</code>	$\geq 99.5\%$	Synthetic check every 5 min from the Lambda canary [<code>SyntheticRunsPassed</code> , summed]	<code>gamehub-prod-synthetic-check-failed</code>
SLO-2	API availability – <code>/api/*</code> answers without a 5xx	$\geq 99.5\%$	nginx access log → metric filter [<code>ServerError</code> vs <code>Requests</code>]	<code>gamehub-prod-http-5xx</code>
SLO-3	API latency – p99 of <code>/api/*</code> request time	≤ 1.5 s	nginx access log <code>request_time</code> → metric filter [<code>ApiRequestTimeSeconds</code>]	<code>gamehub-prod-api-latency-p99</code>
SLO-4	Host health – instance status checks pass	no failing check for 2 min	EC2 <code>StatusCheckFailed</code>	<code>gamehub-prod-instance-status-check-failed</code>
SLO-5	Host capacity – CPU, memory and disk stay off the ceiling	CPU < 85 % · memory < 90 % · disk < 85 %	EC2 metrics + CloudWatch agent [<code>CWAgent</code>]	<code>gamehub-prod-cpu-high</code> , <code>gamehub-prod-memory-high</code> , <code>gamehub-prod-disk-high</code> , <code>gamehub-prod-cpu-credit-balance-low</code>

Two things are intentionally **not** SLOs:

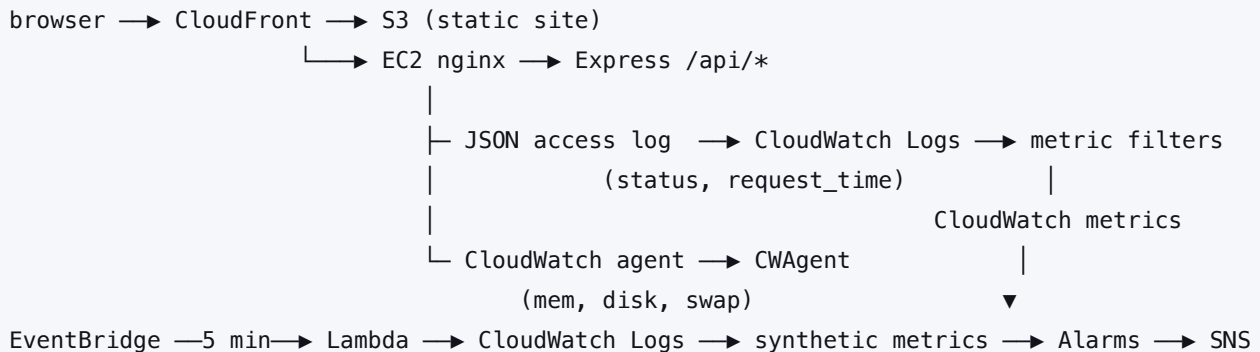
- **Game frame rate** and client-side errors – browser-only metrics, not observable from the server, so claiming a number for them would be theatre.
- **Write durability of scores** – the MySQL volume lives on the instance's EBS root volume. A single-instance topology cannot honestly promise more than "the snapshot schedule in `infra/backend.yaml`".

2. Error budgets

Objective	Budget per 30 days	What "burning" it means
SLO-1 / SLO-2 availability $\geq 99.5\%$	3 h 39 m of degraded service	One failed 5-minute synthetic check costs 0.03 % of the budget – 36 consecutive failures exhaust it
SLO-3 latency p99 ≤ 1.5 s	3 h 39 m of p99 above target	The alarm needs 2 consecutive 5-minute windows, so a single slow render is not a page

The budget is not a dashboard number to admire; it decides **priority**. While the budget is intact, new features ship. Once an alarm page is unexplained, the availability work comes first – which for this project means the next roadmap item is postponed, not parallelised.

3. How the numbers are actually produced



- Availability and latency come from **measured traffic and a synthetic check**, never from the alarm configuration.
- `Requests` counts every log line, `ServerError` counts `status >= 500`; the error ratio is a CloudWatch **metric math** expression on the dashboard `[ ServerErrors / Requests ]`, not a metric filter, so neither side can drift.
- The canary emits **one JSON line per run** and the alarm sums it: `1` means "the run happened and both checks passed". A metric filter publishes its default value `[0]` for every log event that does not match – including the Lambda's own `START` / `END` / `REPORT` lines – so summing is the only statistic that ignores the noise instead of averaging it in.
- Latency is the nginx `$request_time`, which includes the upstream Express call – the number a user actually feels, not just the time spent in Node.
- `TreatMissingData` is `breaching` only for the two availability alarms. For the synthetic check, silence means the checker itself died, which is a page.

4. When an objective is missed

1. The alarm fires to the `gamehub-prod-alarms` SNS topic (email subscription).
2. The on-call response is written down in `runbook.md` – one section per alarm, with the exact commands to look at, not advice to "investigate".
3. If the miss was caused by a deploy, the rollback is `git revert` on `main`, which re-triggers the pipeline and republishes the previous commit. The container images are pinned to commits, so the previous version is one revert away.
4. If the miss was not caused by a deploy, the fix is a code or infra change on a branch, reviewed and merged like any other change – no hand-editing in the AWS console.