The Device Abstraction: Deep Dive

> *Last updated: 2026-02-27. Based on full codebase audit of all device-related source files.*

Table of Contents

---

1. What the Abstraction Provides

The Device abstraction answers one question: **"Where did this happen?"** Every event, session, and git activity row is attributed to a device — the physical or virtual machine where the coding occurred.

A device is a self-registered entity. There's no admin panel, no device provisioning API, no enrollment flow. A machine becomes a device when `fuel-code init` runs and generates a ULID, stored in `~/.fuel-code/config.yaml` . From that point, every event emitted from that machine carries the device ID, and the backend upserts the device record on first contact.

Core value proposition: Attribution and cross-device awareness. "This session ran on my MacBook," "these commits came from the remote dev env," "I've been active on 2 machines this week."

The Device is **one of the original five CORE.md abstractions** (Workspace, Session, Event, Device, Blueprint). It's described in the abstractions audit as "functionally complete" with a minor type/DB column name mismatch.

---

2. The Identity Model

2.1 No Fingerprinting

Device IDs are **pure ULIDs** (Universally Unique Lexicographically Sortable Identifiers). They are NOT hardware fingerprints, NOT MAC address hashes, NOT machine-specific derivations. A ULID is generated once during `fuel-code init` and stored locally. If you delete `~/.fuel-code/config.yaml` and re-init, you get a **new device** — the old one becomes an orphan in the database.

Why no fingerprinting?

- Hardware fingerprints are fragile (change on OS upgrade, VM migration, etc.)
- They prevent device portability (moving config to a new machine)
- They create privacy concerns (tracking across reinstalls)
- ULIDs are simple, collision-free, and sort by time

2.2 ULID Format

Generated by `packages/shared/src/ulid.ts` via the `ulidx` library:

```
01HXYZ1234567890ABCDEF  (26-char Crockford Base32)
├──────────┤├──────────┤
  10-char       16-char
 timestamp      random
 (48-bit)
```

The timestamp component means device IDs sort chronologically — a device created today sorts after one created yesterday. This is useful for "ORDER BY id" queries.

2.3 Local Storage

`~/.fuel-code/config.yaml`:

```yaml
# fuel-code CLI configuration
# Generated by 'fuel-code init'. Edit with care.

backend:
  url: https://api.fuel-code.dev
  api_key: sk-abc123
device:
  id: 01HXYZ1234567890ABCDEF
  name: Johns-MacBook-Pro
  type: local
pipeline:
  queue_path: /Users/john/.fuel-code/queue
  drain_interval_seconds: 10
  batch_size: 50
  post_timeout_ms: 5000
```

The config file is written with mode `0o600` (owner read/write only) because it contains the API key. The write is atomic (temp file + rename).

---

3. The Database Schema

3.1 `devices` Table

`packages/server/src/db/migrations/001_initial.sql`:

```sql
CREATE TABLE IF NOT EXISTS devices (
    id              TEXT PRIMARY KEY,
    name            TEXT NOT NULL,
    type            TEXT NOT NULL CHECK (type IN ('local', 'remote')),
    hostname        TEXT,
    os              TEXT,
    arch            TEXT,
    status          TEXT NOT NULL DEFAULT 'online'
                    CHECK (status IN ('online', 'offline', 'provisioning',
```

```
'terminated')),
    metadata        JSONB NOT NULL DEFAULT '{}',
    first_seen_at   TIMESTAMPTZ NOT NULL DEFAULT now(),
    last_seen_at    TIMESTAMPTZ NOT NULL DEFAULT now()
);
```

Column details:

Column	Type	Nullable	Purpose
`id`	TEXT PK	NO	ULID from `fuel-code init`. Client-generated, NOT server-generated.
`name`	TEXT	NO	Human-readable name. Defaults to `os.hostname()`, user-overridable via `--name`.
`type`	TEXT	NO	`"local"` (laptop/desktop) or `"remote"` (EC2, codespace). Constrained by CHECK.
`hostname`	TEXT	YES	The OS hostname string. May differ from `name` (name is editable, hostname is auto-detected).
`os`	TEXT	YES	Platform string: `"darwin"`, `"linux"`, `"win32"`. From `process.platform`.
`arch`	TEXT	YES	Architecture: `"arm64"`, `"x86_64"`. From `process.arch`.
`status`	TEXT	NO	Current operational state. 4-value enum (see Section 7).
`metadata`	JSONB	NO	Extensible. Currently always `{}` — reserved for CPU, memory, etc.
`first_seen_at`	TIMESTAMPTZ	NO	Immutable. When the device first contacted the backend.
`last_seen_at`	TIMESTAMPTZ	NO	Updated on every event. The "heartbeat" signal.

Indexes: No dedicated indexes on the `devices` table. Two related indexes exist on other tables:

- `idx_sessions_device` on `sessions(device_id, started_at DESC)` — list sessions by device
- `idx_events_device` on `events(device_id, timestamp DESC)` — list events by device

3.2 TypeScript Type

`packages/shared/src/types/device.ts`:

```typescript
export type DeviceType = "local" | "remote";
export type DeviceStatus = "online" | "offline" | "provisioning" | "terminated";

export interface Device {
  id: string;                          // ULID primary key
  type: DeviceType;                    // "local" or "remote"
  name: string;                        // Human-readable name
```

```
  status: DeviceStatus;                    // Current status
  platform: string;                        // OS/platform info
  os_version: string;                      // OS version string
  metadata: Record<string, unknown>;       // Arbitrary metadata
  first_seen_at: string;                   // ISO-8601 timestamp
  last_seen_at: string;                    // ISO-8601 timestamp
}
```

3.3 Type/DB Column Name Mismatch

This is the **known drift issue** for the Device abstraction:

TypeScript Field	Database Column	Notes
`id`	`id`	Aligned
`type`	`type`	Aligned
`name`	`name`	Aligned
`status`	`status`	Aligned
`platform`	`os`	**MISMATCHED**: Type says `platform`, DB says `os`
`os_version`	*(no column)*	**MISSING**: Type defines it, DB has no such column
(no field)	`hostname`	**MISSING**: DB has it, type omits it
(no field)	`arch`	**MISSING**: DB has it, type omits it
`metadata`	`metadata`	Aligned
`first_seen_at`	`first_seen_at`	Aligned
`last_seen_at`	`last_seen_at`	Aligned

Impact: Code that casts a DB row to `Device` and reads `device.platform` gets `undefined` (the column is named `os`). Code reading `device.os_version` also gets `undefined` (no such column). The `hostname` and `arch` columns exist in the DB but have no TypeScript representation.

This drift is **partially invisible** because:

1. The resolver writes to the correct DB columns (`hostname`, `os`, `arch`)
2. The API route does `SELECT *` and returns all columns, which includes the correct names
3. The CLI reads from the API response, so it gets `hostname`, `os`, `arch` (correct)
4. Only code that explicitly uses the `Device` interface would get wrong field names

---

4. Device Initialization (`fuel-code init`)

`packages/cli/src/commands/init.ts`:

4.1 The Init Flow

```
fuel-code init [--name NAME] [--url URL] [--api-key KEY] [--force]
     │
     ├─ Config exists and no --force?
     │     → "Already initialized. Use --force to re-initialize."
     │
     ├─ --force with valid existing config?
     │     → PRESERVE existing device.id (critical: doesn't orphan the device)
     │
     ├─ --force with corrupted config, or fresh install?
     │     → Generate new ULID
     │
     ├─ Determine device name:
     │     --name flag > os.hostname()
     │     Validate: /^[a-zA-Z0-9._-]{1,64}$/
     │
     ├─ Resolve backend URL:
     │     --url > FUEL_CODE_BACKEND_URL env var > error
     │
     ├─ Resolve API key:
     │     --api-key > FUEL_CODE_API_KEY env var > error
     │
     ├─ Build config object:
     │     device: { id: ULID, name: hostname, type: "local" }
     │     pipeline: { queue_path, drain_interval: 10s, batch_size: 50, timeout: 5s }
     │
     ├─ saveConfig(config)  ← atomic write (tmp + rename), mode 0o600
     │
     ├─ Test connectivity: GET {url}/api/health (5s timeout, warn-only)
     │
     └─ Auto-trigger: scan for historical CC sessions, spawn background backfill
```

4.2 Key Design Decisions

ID preservation on `--force`: If a user runs `fuel-code init --force` to update their backend URL, the existing device ID is preserved. This prevents creating an orphan device record in the database. Only a corrupted/invalid config causes a fresh ID generation.

Device type always `"local"`: The `init` command hardcodes `type: "local"`. Remote devices would be initialized differently during EC2 provisioning (Phase 5, not yet implemented).

Name validation: Device names must match `[a-zA-Z0-9._-]{1,64}`. This prevents spaces, unicode, and special characters that could break shell scripts or database queries.

Automatic backfill: After init, the command scans `~/.claude/projects/` for historical CC sessions and spawns a detached `fuel-code backfill` process. This means the first init populates the database with past session data without blocking.

---

5. The Resolution Pipeline

`packages/core/src/device-resolver.ts`:

5.1 `resolveOrCreateDevice(sql, deviceId, hints?)`

```
export async function resolveOrCreateDevice(
  sql: Sql,
  deviceId: string,
  hints?: {
    name?: string;
    type?: DeviceType;
    hostname?: string;
    os?: string;
    arch?: string;
  },
): Promise<string> {
  const name = hints?.name || "unknown-device";
  const type = hints?.type || "local";

  const [row] = await sql`
    INSERT INTO devices (id, name, type, hostname, os, arch, metadata)
    VALUES (${deviceId}, ${name}, ${type}, ${hints?.hostname ?? null}, ${hints?.os
?? null}, ${hints?.arch ?? null}, ${JSON.stringify({})})
    ON CONFLICT (id) DO UPDATE SET
      last_seen_at = now(),
      name = CASE WHEN devices.name = 'unknown-device' AND ${name} != 'unknown-
device' THEN ${name} ELSE devices.name END,
      hostname = COALESCE(EXCLUDED.hostname, devices.hostname),
      os = COALESCE(EXCLUDED.os, devices.os),
      arch = COALESCE(EXCLUDED.arch, devices.arch)
    RETURNING id
  `;

  return row.id;
}
```

Upsert semantics (called on every event):

Scenario	Behavior
New device	INSERT with hints. Missing fields stay NULL.
Existing device, same hints	UPDATE `last_seen_at` only.
Existing device, better name	If current name is `"unknown-device"` and hints provide a real name, upgrade it.
Existing device, new hostname/os/arch	Fill in NULLs via `COALESCE`. Don't overwrite existing values.
Existing device, different name	Keep the existing name (first-seen wins, unless it was "unknown-device").

Why the `"unknown-device"` heuristic? Events can arrive before the device has been properly initialized (e.g., from a git hook before `fuel-code init` completes). These events create a device with name

`"unknown-device"`. When the next event arrives with the real hostname in the hints, the name is upgraded.

5.2 `updateDeviceLastSeen(sql, deviceId)`

```
export async function updateDeviceLastSeen(
  sql: Sql,
  deviceId: string,
): Promise<void> {
  await sql`
    UPDATE devices SET last_seen_at = now() WHERE id = ${deviceId}
  `;
}
```

A lightweight heartbeat function. Currently not called directly — `last_seen_at` is updated by the upsert in `resolveOrCreateDevice()`. This function exists for future periodic heartbeat events (`system.heartbeat`).

---

6. The Hints Transport Mechanism

Device metadata (name, type, hostname, os, arch) travels from the CLI to the backend via a **transport hints** pattern:

6.1 Injection (CLI Side)

`packages/cli/src/commands/emit.ts`:

```
// Inject device hints into event.data before POST
if (config?.device.name) {
  data._device_name = config.device.name;
  data._device_type = config.device.type;
}
```

The CLI reads `device.name` and `device.type` from `~/.fuel-code/config.yaml` and injects them as underscore-prefixed fields into the event's `data` payload.

6.2 Extraction (Backend Side)

`packages/core/src/event-processor.ts`:

```
// Extract hints from event.data
const deviceHints = event.data._device_name
  ? {
      name: event.data._device_name as string,
      type: (event.data._device_type as "local" | "remote") ?? "local",
    }
  : undefined;

// Resolve device with hints
await resolveOrCreateDevice(sql, event.device_id, deviceHints);
```

```
// Strip hints before persisting the event
delete event.data._device_name;
delete event.data._device_type;
```

6.3 Design Analysis

This is an **untyped mutation of the event payload** — a pragmatic hack that works but has implications:

Aspect	Assessment
Works correctly	Yes — device names propagate from client to server
Type-safe	No — `_device_name` and `_device_type` are not in any TypeScript type
Mutates event data	Yes — `delete` modifies the event object in-place
Leaks into persistence?	No — hints are stripped before the INSERT INTO events
Violates immutability?	Technically yes — events should be immutable facts
Alternative	Could use HTTP headers or a wrapper envelope

The pattern works because:

1. Hints are injected late (just before POST)
2. They're extracted early (before handler dispatch)
3. They're stripped before event persistence
4. The underscore prefix signals "internal, don't rely on this"

6.4 What's NOT Transported

Only `name` and `type` are sent as hints. The `hostname`, `os`, and `arch` fields are **never populated from the CLI side**. The resolver defaults them to NULL, and they stay NULL unless manually set.

This means for most devices:

```
-- Typical device row after many events
id:         '01HXYZ...'
name:       'Johns-MacBook-Pro'
type:       'local'
hostname:   NULL
os:         NULL
arch:       NULL
status:     'online'
metadata:   '{}'
```

The `hostname`, `os`, and `arch` columns exist in the schema but are never populated in practice. They would be populated by a proper device registration endpoint (not yet implemented) or by extending the hints mechanism.

---

7. Status and Lifecycle

7.1 Status States

```
CHECK (status IN ('online', 'offline', 'provisioning', 'terminated'))
```

Status	Meaning	When Set
`online`	Device is active (events arriving recently)	Default on INSERT
`offline`	Device has gone quiet	Not set by any code
`provisioning`	Remote device being set up	Phase 5 (remote envs, not implemented)
`terminated`	Remote device shut down	Phase 5 (remote envs, not implemented)

7.2 Status is Effectively Static

No code in the codebase transitions device status. Every device starts as `"online"` and stays `"online"` forever. The `offline`, `provisioning`, and `terminated` states are forward declarations for Phase 5 (remote environments).

There is no:

- Heartbeat timeout that flips `online` → `offline`
- Cron job or background process that checks device liveness
- Explicit `system.heartbeat` event type handler
- API endpoint to set device status

Liveness inference: The `last_seen_at` timestamp is the only signal. A client could determine "is this device offline?" by checking `now() - last_seen_at > threshold`, but this is not implemented anywhere.

7.3 Remote Device Lifecycle (Planned)

Per CORE.md, remote devices would follow:

```
provisioning → online → offline → terminated
```

With EC2 instances:

- `provisioning`: Instance launched, Docker container building
- `online`: Dev environment ready, sessions running
- `offline`: Instance idle, no active sessions
- `terminated`: Instance destroyed, resources released

None of this is implemented. The `remote_envs` table doesn't exist.

---

8. Query Surfaces (API + CLI)

8.1 API Endpoints

`GET /api/devices` — List all devices with aggregates

`packages/server/src/routes/devices.ts`:

No pagination (single-user system, small device count). Uses CTEs to avoid cross-join inflation:

```
WITH device_sessions AS (
  SELECT s.device_id,
    COUNT(*)::int AS session_count,
    COUNT(CASE WHEN s.lifecycle IN ('detected', 'capturing') THEN 1 END)::int AS
active_session_count,
    MAX(s.started_at) AS last_session_at,
    COALESCE(SUM(s.cost_estimate_usd), 0) AS total_cost_usd,
    COALESCE(SUM(s.duration_ms), 0) AS total_duration_ms
  FROM sessions s
  GROUP BY s.device_id
),
device_workspaces AS (
  SELECT wd.device_id,
    COUNT(DISTINCT wd.workspace_id)::int AS workspace_count
  FROM workspace_devices wd
  GROUP BY wd.device_id
)
SELECT d.*,
  COALESCE(ds.session_count, 0) AS session_count,
  COALESCE(dw.workspace_count, 0) AS workspace_count,
  COALESCE(ds.active_session_count, 0) AS active_session_count,
  ds.last_session_at,
  COALESCE(ds.total_cost_usd, 0) AS total_cost_usd,
  COALESCE(ds.total_duration_ms, 0) AS total_duration_ms
FROM devices d
LEFT JOIN device_sessions ds ON ds.device_id = d.id
LEFT JOIN device_workspaces dw ON dw.device_id = d.id
ORDER BY d.last_seen_at DESC
```

Why CTEs? A flat `JOIN sessions JOIN workspace_devices` produces N*M rows per device (sessions × workspaces), inflating all aggregates. Pre-aggregating each dimension in its own CTE keeps counts correct.

Response:


```
{
  "devices": [
    {
      "id": "01HXYZ...",
      "name": "Johns-MacBook-Pro",
      "type": "local",
      "hostname": null,
      "os": null,
      "arch": null,
      "status": "online",
      "metadata": {},
      "first_seen_at": "2026-02-01T00:00:00Z",
      "last_seen_at": "2026-02-27T12:00:00Z",
      "session_count": 42,
      "workspace_count": 3,
```

```
      "active_session_count": 1,
      "last_session_at": "2026-02-27T12:00:00Z",
      "total_cost_usd": 3.14,
      "total_duration_ms": 360000
    }
  ]
}
```

`GET /api/devices/:id` — Device detail

Lookup by ULID only (no fuzzy matching at the API level). Returns 404 if not found.

Two parallel queries:

1. **Workspaces**: Via `workspace_devices` junction — returns workspace info + `local_path`, `hooks_installed`, `git_hooks_installed`, `last_active_at`
2. **Recent sessions**: Last 10 sessions on this device, with workspace names joined

Response:

```
{
  "device": { /* full device record */ },
  "workspaces": [
    {
      "id": "01HABC...",
      "canonical_id": "github.com/user/repo",
      "display_name": "repo",
      "default_branch": "main",
      "local_path": "/Users/john/Desktop/fuel-code",
      "hooks_installed": false,
      "git_hooks_installed": true,
      "last_active_at": "2026-02-27T12:00:00Z"
    }
  ],
  "recent_sessions": [
    {
      "id": "01HSES...",
      "workspace_id": "01HABC...",
      "lifecycle": "summarized",
      "started_at": "2026-02-27T12:00:00Z",
      "workspace_name": "fuel-code",
      "workspace_canonical_id": "github.com/user/fuel-code"
    }
  ]
}
```

8.2 CLI Commands

`fuel-code status` — Device info (local only)

`packages/cli/src/commands/status.ts`:

Reads from `~/.fuel-code/config.yaml` directly — no backend call needed:

```
Device:
  ID:    01HXYZ1234567890ABCDEF
  Name:  Johns-MacBook-Pro
  Type:  local
```

`fuel-code sessions --device <name>` — Filter sessions by device

`packages/cli/src/commands/sessions.ts` :

The sessions list command includes a `DEVICE` column and supports `--device` filtering:

```
ID         STATUS       WORKSPACE   DEVICE           DURATION  STARTED
01HSES1    summarized   fuel-code   Johns-MacBook    15m       2m ago
01HSES2    summarized   fuel-code   Johns-MacBook    30m       1h ago
```

Device name resolution uses `resolveDeviceName(api, nameOrId)` (see Section 9).

No dedicated `fuel-code devices` or `fuel-code device <name>` commands exist. Device information is accessed via `fuel-code status` (local), `fuel-code workspace <name>` (shows linked devices), or the API directly.

---

9. Name Resolution (CLI Fuzzy Matching)

`packages/cli/src/lib/resolvers.ts` :

```
export async function resolveDeviceName(
  api: FuelApiClient,
  nameOrId: string,
): Promise<string> {
  // 1. ULID? → return as-is
  if (/^[0-9A-Z]{26}$/.test(nameOrId)) return nameOrId;

  // 2. Fetch all devices
  const devices = await api.listDevices();

  // 3. Exact match on name (case-insensitive)
  const exact = devices.find(d => d.name.toLowerCase() === nameOrId.toLowerCase());
  if (exact) return exact.id;

  // 4. Single prefix match
  const prefixMatches = devices.filter(d =>
d.name.toLowerCase().startsWith(nameOrId.toLowerCase()));
  if (prefixMatches.length === 1) return prefixMatches[0].id;

  // 5. Ambiguous
  if (prefixMatches.length > 1) throw new ApiError(`Ambiguous: ${names}`, 400);

  // 6. Not found
  throw new ApiError(`Device "${nameOrId}" not found. Available: ${available}`,
404);
}
```

Same cascade as workspace resolution: ULID passthrough → exact match → prefix match → error with suggestions.

---

10. Cross-Abstraction Relationships

10.1 Device ↔ Workspace (Many-to-Many)

Via `workspace_devices` junction table. A device can have multiple repos checked out. A workspace (repo) can be checked out on multiple devices.

```
Device: Johns-MacBook-Pro
  ├── Workspace: fuel-code     (local_path: /Users/john/Desktop/fuel-code)
  ├── Workspace: other-project (local_path: /Users/john/Projects/other)
  └── Workspace: _unassociated

Device: remote-abc
  └── Workspace: fuel-code     (local_path: /home/deploy/fuel-code)
```

10.2 Device ↔ Session (One-to-Many)

Every session has `device_id TEXT NOT NULL REFERENCES devices(id)`. A session runs on exactly one device. A device can have many sessions (sequential, rarely parallel).

10.3 Device ↔ Event (One-to-Many)

Every event has `device_id TEXT NOT NULL REFERENCES devices(id)`. Events are attributed to their originating device.

10.4 Device ↔ Git Activity (Indirect)

Git activity rows have `device_id TEXT NOT NULL REFERENCES devices(id)`. Combined with `workspace_id` and `session_id`, this enables queries like "commits from device X on workspace Y during session Z."

10.5 Device in Git Session Correlation

The git-session correlator (`packages/core/src/git-correlator.ts`) matches on `workspace_id + device_id + lifecycle + timestamp`. The device_id is critical — it prevents a session on device A from claiming a commit made on device B (even if both devices work on the same repo).

10.6 Device in Redis Consumer

`packages/server/src/redis/stream.ts` uses `os.hostname()` as part of the Redis consumer name:

```
const consumerName = `consumer-${os.hostname()}-${process.pid}`;
```

This is **not the fuel-code device ID** — it's the raw hostname, used for Redis consumer group identification. Multiple fuel-code server processes on the same host get unique names via PID.

---

11. Gap Analysis: What's Missing or Misaligned

Gap 1: Type/DB Column Name Mismatch

Severity: Medium

The `Device` TypeScript interface uses `platform` and `os_version` for fields that are named `os` and `arch` (respectively... sort of) in the database. The type also defines `os_version` which has no corresponding column, and omits `hostname` and `arch` which are real columns.

TypeScript	Database	Status
`platform`	`os`	Name mismatch
`os_version`	*(none)*	Phantom field
(none)	`hostname`	Missing from type
(none)	`arch`	Missing from type

Impact: Any code using the `Device` interface to read DB rows gets wrong field names. In practice this is mitigated because the API route does `SELECT *` and the CLI reads from the API response (which uses DB column names), but it's a trap for future code that trusts the interface.

Gap 2: `hostname`, `os`, `arch` Never Populated

Severity: Medium

The CLI emit command only sends `_device_name` and `_device_type` as hints. It does NOT send `hostname`, `os`, or `arch`. The device resolver accepts these as optional hints but they're never provided.

Result: Every device has NULL for `hostname`, `os`, and `arch` — three columns that exist in the schema but are permanently empty.

Potential fix: Extend the emit command to include `os.hostname()`, `process.platform`, and `process.arch` as additional hints:

```
data._device_hostname = os.hostname();
data._device_os = process.platform;
data._device_arch = process.arch;
```

Gap 3: No Status Transitions

Severity: Medium

The `status` column has 4 valid values but only `"online"` is ever set. No code transitions devices to `"offline"`, `"provisioning"`, or `"terminated"`.

There's no:

- Background job that marks stale devices as offline
- API endpoint to change device status
- Heartbeat event type that refreshes status
- TTL-based offline detection

Impact: The device list always shows every device as "online" even if it hasn't sent an event in weeks. The `last_seen_at` timestamp is the only liveness signal.

Potential fix: Implement `system.heartbeat` events emitted periodically by the CLI, plus a background job that transitions `online → offline` when `last_seen_at > threshold`.

Gap 4: No Device Management

Severity: Low

There are no endpoints or CLI commands to:

- Rename a device (update `name`)
- Delete/decommission a device
- View device details via CLI (`fuel-code device <name>`)
- Merge devices (e.g., if a machine was re-initialized with a new ULID)

Devices can only be created implicitly via event processing.

Gap 5: `metadata` Always Empty

Severity: Low

The `metadata` JSONB column is always `'{}'`. The resolver hardcodes it:

```
${JSON.stringify({})}
```

No code path populates device metadata (CPU cores, memory, disk, etc.). The column exists for future use but is currently dead weight.

Gap 6: Remote Device Support Unimplemented

Severity: Low (expected — Phase 5)

The schema supports `type: "remote"` and statuses `provisioning` / `terminated`, but:

- No `remote_envs` table exists
- No EC2 provisioning code exists
- No Blueprint detection exists
- `sessions.remote_env_id` column exists but references nothing

This is expected (Phase 5 scope) but means half the Device abstraction's design surface is unrealized.

Gap 7: No Dedicated CLI Commands

Severity: Low

Unlike workspaces (which have `fuel-code workspaces` and `fuel-code workspace <name>`), devices have no dedicated list or detail commands. Device info is scattered across:

- `fuel-code status` — local config only
- `fuel-code workspace <name>` — shows linked devices
- `fuel-code sessions --device <name>` — filter by device
- API direct access

Potential fix: Add `fuel-code devices` (list with aggregates) and `fuel-code device <name>` (detail with workspaces and recent sessions).

Summary Table

Gap	Severity	Impact	Fix Complexity
Type/DB name mismatch	Medium	Wrong field reads from interface	Low (rename fields)
hostname/os/arch empty	Medium	3 useless columns	Low (extend hints)
No status transitions	Medium	Stale "online" status	Medium (heartbeat system)
No device management	Low	Can't rename/delete devices	Medium
metadata always empty	Low	Dead weight column	Low (populate from hints)
Remote support absent	Low	Expected (Phase 5)	High (entire subsystem)
No dedicated CLI	Low	Info scattered	Low (add commands)

---

12. References

#	File	Description
1	`packages/shared/src/types/device.ts`	TypeScript type: `Device` `DeviceType`, `DeviceStatu`
2	`packages/shared/src/ulid.ts`	ULID generation via `generateId()`
3	`packages/core/src/device-resolver.ts`	`resolveOrCreateDevice(` `updateDeviceLastSeen()`
4	`packages/core/src/workspace-device-link.ts`	`ensureWorkspaceDeviceL` — junction table upsert
5	`packages/core/src/event-processor.ts`	Event processing pipeli — device resolution + h stripping
6	`packages/core/src/git-correlator.ts`	Session correlation — u `device_id` for matching
7	`packages/server/src/db/migrations/001_initial.sql`	Database schema: `devi` `workspace_devices` table
8	`packages/server/src/db/migrations/004_add_git_hooks_prompt_columns.sql`	Hook prompt columns `workspace_devices`
9	`packages/server/src/routes/devices.ts`	API: `GET /api/devices` (l `GET /api/devices/:id` (detail)
10	`packages/cli/src/commands/init.ts`	`fuel-code init` — devic generation, config crea

11	`packages/cli/src/commands/status.ts`	`fuel-code status` — loca device info display
12	`packages/cli/src/commands/emit.ts`	Hint injection: `_device_ _device_type`
13	`packages/cli/src/lib/config.ts`	Config file managemen `loadConfig()`, `saveConfi`
14	`packages/cli/src/lib/resolvers.ts`	`resolveDeviceName()` — fuzzy name/ULID resol
15	`packages/cli/src/lib/api-client.ts`	API client: `listDevices( getDevice()`
16	`packages/server/src/redis/stream.ts`	Redis consumer naming (uses `os.hostname()`, no device ID)