The Blueprint Abstraction: Deep Dive (Planned — Phase 5)

> *Last updated: 2026-02-27. Based on CORE.md spec, database schema design, forward declarations across the codebase, and Phase 5 task definitions. Blueprint is fully spec'd but **not yet implemented**.*

Table of Contents

---

1. What the Abstraction Will Provide

The Blueprint is a **recipe for creating a remote development environment**. It describes what a project needs to run: runtime, dependencies, system packages, Docker configuration, cloud resources, environment variables, ports, and setup commands. It lives as a `.fuel-code/env.yaml` file committed to the repository.

The Blueprint is **one of the original five CORE.md abstractions** (Workspace, Session, Event, Device, Blueprint). CORE.md describes it as "a recipe for creating a remote Device. The `.fuel-code/env.yaml` file."

Core value proposition: Different projects need radically different environments. A Python ML project needs CUDA drivers and a GPU instance. A Node API needs nothing special. The Blueprint captures this per-workspace in a human-reviewable, version-controlled format. It enables `fuel-code remote up` — a single command to spin up a disposable EC2 instance with Docker, the project's code, all dependencies, Claude Code, and fuel-code hooks pre-installed.

Why it matters for the system: Without Blueprints, the "remote" dimension of fuel-code is purely observational (tracking existing remote sessions). With Blueprints, fuel-code becomes an active provisioner — it doesn't just track development activity, it provides the environments where that activity happens.

---

2. Current Status: What Exists Today

Blueprint is the only one of the five core abstractions with **zero implementation** in the codebase. However, several forward declarations and placeholders exist:

What Exists	Where	Purpose
4 remote event types	`shared/src/types/event.ts`	`remote.provision.start`, `remote.provision.ready`, `remote.provision.error`, `remote.terminate`
`remote_env_id` column	`sessions` table	Nullable TEXT column with comment: "FK to remote_envs added in Phase 5 migration"
`remote` device type	`devices` table	`CHECK (type IN ('local', 'remote'))` — remote devices are a first-class concept
Device status states	`devices` table	`provisioning` and `terminated` statuses exist alongside `online` and `offline`
WebSocket broadcast	`server/src/ws/websocket.ts`	`broadcastRemoteUpdate()` function already defined
WsClient handler	Client-side WebSocket	`remote.update` message type already handled
Phase 5 task files	`tasks/phase-5/`	15 task files defining the full implementation plan
CORE.md spec	`tasks/CORE.md`	Full schema, provisioning flow, CLI commands, env.yaml format

The system was designed from day one with Blueprints in mind. The "remote" concept permeates the existing abstractions, but the actual provisioning and env.yaml logic is Phase 5.

---

3. The env.yaml Format

The Blueprint is materialized as a `.fuel-code/env.yaml` file at the root of a workspace:

```
# .fuel-code/env.yaml — auto-generated, human-reviewed, committed to repo
runtime: node
version: "22"
package_manager: bun

system_deps:
  - postgresql-client
  - redis-tools

docker:
  base_image: "node:22-bookworm"
  additional_packages: []

resources:
  instance_type: t3.xlarge
  region: us-east-1
  disk_gb: 50
```

```
environment:
  NODE_ENV: development

ports:
  - 3000
  - 5432

setup:
  - bun install
```

Field Semantics

Field	Type	Purpose
`runtime`	string	Primary language runtime: `node`, `python`, `rust`, `go`, etc.
`version`	string	Runtime version. Quoted to prevent YAML float coercion ("22" not 22).
`package_manager`	string	Package manager: `bun`, `npm`, `yarn`, `pnpm`, `pip`, `uv`, `cargo`
`system_deps`	string[]	APT packages to install in the Docker container
`docker.base_image`	string	Docker Hub image to use as the base
`docker.additional_packages`	string[]	Extra Docker-level packages
`resources.instance_type`	string	EC2 instance type
`resources.region`	string	AWS region
`resources.disk_gb`	number	EBS volume size
`environment`	map	Environment variables injected into the container
`ports`	number[]	Ports to expose (mapped from container to host)
`setup`	string[]	Commands run after container start (install deps, build, etc.)

Design Principles

- **Auto-detected with human review**: The system generates env.yaml from repo analysis (package.json, Dockerfile, pyproject.toml, etc.), but the user reviews and can edit before committing.
- **Committed to the repo**: This is intentional — the Blueprint is version-controlled and shared. If a teammate clones the repo, they get the same environment spec.
- **Frozen on provision**: When `fuel-code remote up` runs, the env.yaml is read, frozen into immutable JSON, and stored in the `blueprints` table and the `remote_envs.blueprint` column. This ensures reproducibility — the provisioned environment matches exactly what was specified, even if env.yaml changes later.

---

4. The Blueprint Lifecycle

```
Scan project repo → Auto-detect from package.json/Dockerfile/etc.
       │
       ▼
Generate .fuel-code/env.yaml → Show to user for review
       │
       ▼
User reviews, edits → Commits to repo
       │
       ▼
fuel-code remote up → Freeze env.yaml → Immutable JSON in blueprints table
       │
       ▼
Provision EC2 + Docker → Clone repo → Run setup commands → Health check
       │
       ▼
Remote device active → Events flow to same backend → Session/git tracking identical
       │
       ▼
Terminate → remote.terminate event → Update remote_envs record → Device status =
terminated
```

The Blueprint itself is stateless — it's just a YAML file. The stateful entity is the **Remote Environment** (`remote_envs` table) which tracks the provisioned instance. A single Blueprint can produce many Remote Environments (spin up, tear down, spin up again).

---

5. The Database Schema (Planned)

From CORE.md spec:

```sql
CREATE TABLE blueprints (
    id              TEXT PRIMARY KEY,        -- ULID
    workspace_id    TEXT REFERENCES workspaces(id), -- null for global/template
blueprints
    name            TEXT NOT NULL,
    source          TEXT NOT NULL,           -- "auto-detected" | "manual"
    detected_from   TEXT,                    -- repo path or URL that was scanned
    config          JSONB NOT NULL,          -- full env.yaml content as JSON
    created_at      TIMESTAMPTZ NOT NULL DEFAULT now(),
    updated_at      TIMESTAMPTZ NOT NULL DEFAULT now()
);
```

Key Design Decisions

- **`workspace_id` is nullable**: Blueprints can be global templates (not tied to a workspace). Example: a "Node 22 with PostgreSQL" template that users can start from.
- **`source` distinguishes origin**: `auto-detected` means the system scanned the repo and generated it. `manual` means the user wrote it by hand or from a template.
- **`config` is JSONB**: The full env.yaml content is stored as JSON, not as a reference to the file. This is the "frozen" snapshot — it captures the exact configuration at the time the Blueprint was registered.

- `detected_from` : Records what was scanned to produce the auto-detection (e.g., the repo path). Useful for auditing: "this Blueprint was generated from analyzing /Users/john/Desktop/fuel-code".

---

6. The Remote Environment Schema (Planned)

From CORE.md spec:

```
CREATE TABLE remote_envs (
    id                TEXT PRIMARY KEY,        -- ULID
    workspace_id      TEXT NOT NULL REFERENCES workspaces(id),
    device_id         TEXT REFERENCES devices(id), -- the EC2 instance's device
identity
    status            TEXT NOT NULL DEFAULT 'provisioning'
                      CHECK (status IN (
                          'provisioning', 'ready', 'active', 'idle', 'terminated',
'error'
                      )),
    instance_id       TEXT,                    -- EC2 instance ID
    instance_type     TEXT NOT NULL,
    region            TEXT NOT NULL,
    public_ip         TEXT,
    ssh_key_s3_key    TEXT,                    -- S3 key for ephemeral SSH key pair
    blueprint         JSONB NOT NULL,          -- snapshot of env.yaml used to
provision
    ttl_minutes       INTEGER NOT NULL DEFAULT 480, -- 8 hours default
    idle_timeout_minutes INTEGER NOT NULL DEFAULT 60,
    cost_per_hour_usd NUMERIC(6, 3),
    total_cost_usd    NUMERIC(8, 3),
    provisioned_at    TIMESTAMPTZ NOT NULL DEFAULT now(),
    ready_at          TIMESTAMPTZ,
    terminated_at     TIMESTAMPTZ,
    termination_reason TEXT,
    metadata          JSONB NOT NULL DEFAULT '{}'
);

CREATE INDEX idx_remote_envs_workspace ON remote_envs(workspace_id);
CREATE INDEX idx_remote_envs_active ON remote_envs(status) WHERE status NOT IN
('terminated', 'error');
```

The remote_envs ↔ sessions FK

The existing `sessions.remote_env_id` column will gain a foreign key constraint once this table is created:

```
ALTER TABLE sessions ADD CONSTRAINT fk_sessions_remote_env
    FOREIGN KEY (remote_env_id) REFERENCES remote_envs(id);
```

Status Lifecycle

```
provisioning → ready → active → idle → terminated
      │                  │        │
      └→ error           └→ idle  └→ terminated
```

Status	Meaning
`provisioning`	EC2 instance launching, Docker building, setup commands running
`ready`	Health check passed, SSH available, waiting for user to connect
`active`	User connected, Claude Code session running
`idle`	No active session, idle timeout counting down
`terminated`	Instance destroyed (manual, TTL, idle timeout, or error)
`error`	Provisioning failed (bad AMI, Docker build error, health check fail)

Cost Tracking

- **`cost_per_hour_usd`** : Based on EC2 instance type pricing
- **`total_cost_usd`** : Accumulated from `provisioned_at` to `terminated_at`
- **`termination_reason`** : `"manual"` | `"ttl"` | `"idle"` | `"error"` — explains why the environment was destroyed

---

7. The Provisioning Flow

From CORE.md spec, `fuel-code remote up` triggers a 4-stage process:

Stage 1: Detect / Load Blueprint (~2s)

1. Check for `.fuel-code/env.yaml` in the current workspace
2. If missing: auto-detect from repo (package.json, Dockerfile, etc.)
3. Generate `.fuel-code/env.yaml` , show to user for review
4. Freeze blueprint → immutable JSON stored in `blueprints` and `remote_envs.blueprint`

Stage 2: Provision EC2 Instance (~45-90s)

1. Generate ephemeral SSH key pair → upload to S3 (`ssh-keys/{remote_env_id}/`)
2. Create/reuse security group (SSH inbound from caller IP only)
3. Launch EC2 instance (Docker-ready AMI) with user-data script:
   - Install Docker
   - Pull Docker image from blueprint
   - Start container with env vars, port mappings
   - Inside container:
     - Clone repo (full, checkout current branch)
     - Run setup commands from blueprint
     - Install fuel-code CLI
     - Run `fuel-code init` (with `device_type=remote`)
     - Install Claude Code
     - Copy user's `~/.claude/` config (settings, permissions, CLAUDE.md)
     - Install CC hooks + git hooks

    - Health check: `claude --version`

4. Tag EC2 instance (`fuel-code:remote-env-id` , `fuel-code:workspace`)
5. Callback to backend: `POST /api/remote/:id/ready`
6. Emit `remote.provision.ready` event

Stage 3: Connect (~1s)

1. Download ephemeral SSH key from S3
2. SSH into EC2 → exec into Docker container
3. User gets a regular terminal (can run `claude` , `git` , etc.)
4. Events flow back to the same backend as local events

Stage 4: Lifecycle

- Auto-terminate after idle timeout (configurable, default 60 min)
- Auto-terminate after TTL (configurable, default 8 hours)
- Manual: `fuel-code remote down <id>`
- On terminate: emit `remote.terminate` event, update `remote_envs` record

---

8. Auto-Detection: How Blueprints Are Born

The auto-detection system will scan the project workspace and infer the correct environment configuration from existing files:

Detection Sources (Planned)

File Detected	Inferences
`package.json`	runtime=node, package_manager from lockfile type, version from engines
`bun.lockb`	package_manager=bun
`yarn.lock`	package_manager=yarn
`pnpm-lock.yaml`	package_manager=pnpm
`pyproject.toml`	runtime=python, version from requires-python
`Pipfile`	runtime=python, package_manager=pipenv
`Cargo.toml`	runtime=rust, version from rust-version
`go.mod`	runtime=go, version from go directive
`Dockerfile`	base_image from FROM, system_deps from RUN apt-get
`docker-compose.yml`	ports from services, environment from env section
`.tool-versions` / `.node-version`	Specific runtime versions

Design Philosophy

- **Inspectable**: The generated env.yaml is a plain file the user can read, understand, and edit

- **Materialized**: Auto-detection produces a concrete file, not a runtime computation. What you see is what you get.
- **Non-destructive**: If env.yaml already exists, auto-detection is skipped. The user's edits are respected.
- **Conservative defaults**: Instance type defaults to `t3.xlarge` (not the cheapest, not the most expensive). Disk defaults to 50GB. TTL defaults to 8 hours. Users can override.

---

9. Remote Device Symmetry

The single most important architectural principle for Blueprints and remote environments:

> *The backend does not distinguish between events from local and remote devices. The processing pipeline is identical.*

Once provisioned, a remote EC2 is just another Device in the topology:

- It has its own `device_id` (generated during `fuel-code init` on the remote)
- fuel-code CLI installed with hooks configured
- Events flow through the same HTTP POST → Redis → Consumer → Processor pipeline
- A `session.start` from `remote-abc` is processed identically to one from `macbook-pro`
- The TUI shows both, distinguished by device name

This means:

- **No remote-specific event processing**: The existing event pipeline handles everything
- **No remote-specific session logic**: Sessions on remote devices go through the same lifecycle
- **No remote-specific transcript handling**: Same S3 upload, same parsing, same summarization
- **Adding a new machine to the topology is just**: "install fuel-code, run init, events flow"

The only remote-specific logic is:

- **Provisioning** (creating the EC2 instance, Docker container, SSH keys)
- **Connection management** (SSH tunneling)
- **Lifecycle management** (idle timeout, TTL, termination)
- **Cost tracking** (per-hour and total cost)

---

10. The 4 Remote Event Types

Already defined in `packages/shared/src/types/event.ts`:

```
| "remote.provision.start"    // EC2 provisioning began
| "remote.provision.ready"    // EC2 + Docker ready, SSH available
| "remote.provision.error"    // Provisioning failed
| "remote.terminate"          // Remote device terminated
```

Payload Schemas (from CORE.md)

```
// remote.provision.ready
interface RemoteProvisionReadyPayload {
  instance_id: string;     // EC2 instance ID
  public_ip: string;       // Public IP for SSH
  ssh_port: number;         // SSH port (usually 22)
  device_id: string;        // The new remote device's ID
}

// remote.terminate
interface RemoteTerminatePayload {
  instance_id: string;
  reason: string;           // "manual" | "ttl" | "idle" | "error"
  uptime_seconds: number;
  total_cost_usd: number | null;
}
```

Event Handler Integration (Planned)

These events will need handlers registered in the `EventHandlerRegistry` :

- `remote.provision.start` : Create `remote_envs` row with status= `provisioning`
- `remote.provision.ready` : Update `remote_envs` with IP, port, device_id; transition status to `ready`
- `remote.provision.error` : Update `remote_envs` with error details; transition status to `error`
- `remote.terminate` : Update `remote_envs` with termination details; transition status to `terminated` ; update device status to `terminated`

---

11. S3 Storage Layout for Remote

From CORE.md:

```
fuel-code-blobs/
├── ssh-keys/
│   └── {remote_env_id}/
│       ├── id_ed25519          # Ephemeral private key
│       └── id_ed25519.pub      # Ephemeral public key
│
└── manifests/
    └── {workspace_canonical_id}/
        └── env.yaml            # Cached environment manifest
```

Key Design Decisions

- **Ephemeral SSH keys**: Generated per-environment, stored in S3, deleted on termination. Not the user's personal SSH keys.
- **Cached manifests**: The frozen env.yaml is stored in S3 alongside the DB record. Belt-and-suspenders for recovery.
- **Keys are per-environment, not per-workspace**: Each `fuel-code remote up` generates fresh keys. No key reuse across environments.

---

12. Forward Declarations: What Already Exists in the Codebase

These are concrete code artifacts that exist today and will be wired up when Blueprint is implemented:

Event Types (shared)

```
// packages/shared/src/types/event.ts
export type EventType =
  | ...
  | "remote.provision.start"
  | "remote.provision.ready"
  | "remote.provision.error"
  | "remote.terminate";
```

These types are part of the EventType union, meaning the event pipeline already accepts and routes them. They just don't have handlers registered yet.

Device Type and Status (database)

```
-- devices table
type TEXT NOT NULL CHECK (type IN ('local', 'remote'))
status TEXT NOT NULL DEFAULT 'online'
    CHECK (status IN ('online', 'offline', 'provisioning', 'terminated'))
```

The device table already supports `remote` as a device type and `provisioning` / `terminated` as statuses. When a remote environment is provisioned, the `fuel-code init` running inside the container will create a device record with `type = 'remote'`.

Session-to-Remote FK (database)

```
-- sessions table
remote_env_id TEXT, -- FK to remote_envs added in Phase 5 migration
```

The nullable column exists. Sessions running on remote devices will have this set to link back to the specific remote environment.

WebSocket Broadcasting (server)

```
// packages/server/src/ws/websocket.ts
broadcastRemoteUpdate() // Already defined, broadcasts to connected WS clients
```

The WebSocket infrastructure already has a function for pushing remote environment status updates to connected clients in real-time.

WS Client Handler (client)

The client-side WebSocket handler already recognizes `remote.update` message types and can display them.

---

13. Cross-Abstraction Relationships (Planned)

```
Blueprint (.fuel-code/env.yaml)
├── belongs_to → Workspace (one env.yaml per workspace)
├── stored_in → blueprints table (frozen snapshot)
├── produces → Remote Environment (many envs from one blueprint)
│
Remote Environment (remote_envs table)
├── belongs_to → Workspace (workspace_id FK)
├── belongs_to → Device (device_id FK — the remote device record)
├── uses → Blueprint (blueprint JSONB — frozen snapshot)
├── has_many → Sessions (sessions.remote_env_id FK)
├── references → S3 (ssh_key_s3_key)
├── generated_by → remote.provision.start Event
├── activated_by → remote.provision.ready Event
├── terminated_by → remote.terminate Event
│
Device (remote variant)
├── type = 'remote'
├── created_by → fuel-code init (running inside the container)
├── emits → same events as local devices (session.*, git.*)
├── linked_from → remote_envs.device_id
```

The Chain: Blueprint → Remote Env → Device → Session

1. User runs `fuel-code remote up` in workspace W
2. Blueprint is read from `.fuel-code/env.yaml`, frozen into JSON
3. EC2 + Docker provisioned → `remote_envs` row created (status=provisioning)
4. Inside container: `fuel-code init` creates a `device` record (type=remote)
5. `remote_envs.device_id` updated to point to the new device
6. User connects via SSH, runs `claude`
7. CC hooks fire → `session.start` event → session row created with `remote_env_id`
8. Same pipeline processes the session: transcript upload, parsing, summary
9. On termination: `remote.terminate` event, `remote_envs.status = terminated`, `device.status = terminated`

---

14. CLI Surface (Planned)

From CORE.md spec:

```
# Blueprint management
fuel-code blueprint show               # Show current workspace's env.yaml
fuel-code blueprint detect             # Auto-detect and generate env.yaml
fuel-code blueprint edit               # Open env.yaml in $EDITOR

# Remote environment management
fuel-code remote up                    # Provision from blueprint
fuel-code remote up --instance t3.2xlarge  # Override instance type
fuel-code remote list                  # List active remote environments
fuel-code remote status <id>           # Remote env details
fuel-code remote ssh <id>              # Connect to remote env
```

```
fuel-code remote down <id>            # Terminate remote env
fuel-code remote down --all           # Terminate all remote envs
```

Expected CLI Behaviors

- `fuel-code remote up` should be a single command with rich TUI output showing provisioning progress
- SSH connection should be transparent — the user drops into a shell inside the Docker container
- Port forwarding should be automatic based on the `ports` section of env.yaml
- `fuel-code remote list` should show status, uptime, cost, workspace, and instance type

---

15. Phase 5 Task Breakdown

The `tasks/phase-5/` directory contains 15 task files defining the full Blueprint implementation. While detailed task content isn't reproduced here, the high-level structure is:

1. **Database migration**: Create `blueprints` and `remote_envs` tables, add FK on sessions
2. **Blueprint types**: TypeScript types for env.yaml schema, Blueprint, RemoteEnv
3. **Blueprint detection**: Auto-scan logic for package.json, Dockerfile, etc.
4. **Blueprint CRUD**: API endpoints and CLI commands for blueprint management
5. **EC2 provisioning**: AWS SDK integration, instance launch, user-data script
6. **SSH key management**: Ephemeral key generation, S3 storage, secure distribution
7. **Docker orchestration**: Container build, setup command execution, health check
8. **Remote event handlers**: Process remote.provision.* and remote.terminate events
9. **SSH connection**: Tunneling, port forwarding, interactive shell
10. **Lifecycle management**: Idle timeout, TTL-based termination, cost tracking
11. **WebSocket integration**: Real-time provisioning progress, status updates
12. **CLI commands**: `fuel-code remote up/down/list/status/ssh`, `fuel-code blueprint show/detect/edit`
13. **Configuration sync**: Copy user's Claude settings to the remote container
14. **Security**: Security group management, key rotation, access control
15. **Testing**: E2E tests for provisioning flow, integration tests for event handlers

---

16. Gap Analysis: Spec vs. Implementation Readiness

1. No AWS SDK Integration (High Priority)

The codebase has S3 integration (via @aws-sdk/client-s3) but no EC2, Security Group, or IAM integration. Phase 5 needs @aws-sdk/client-ec2 and potentially @aws-sdk/client-iam.

2. No Docker Build Pipeline (High Priority)

There's no mechanism to build Docker images from a Blueprint. The spec assumes a base image from Docker Hub with setup commands, but real-world projects may need custom Dockerfiles. The relationship between `docker.base_image` and the project's own Dockerfile (if any) needs clarification.

3. No SSH Tunneling Infrastructure (High Priority)

No code exists for SSH connection management, port forwarding, or interactive shell session piping. This is a significant implementation effort.

4. No Cost Estimation Logic (Medium Priority)

The spec includes `cost_per_hour_usd` and `total_cost_usd` fields but no pricing data or calculation logic. EC2 pricing varies by region and instance type and changes over time.

5. No Idle Detection Mechanism (Medium Priority)

The spec says "auto-terminate after idle timeout" but there's no mechanism to detect idle state. Possible approaches: poll for active SSH connections, check for active CC processes, use CloudWatch metrics.

6. Blueprint Validation Not Specified (Medium Priority)

The env.yaml format is described but there's no Zod schema for validating it. Invalid configurations (non-existent instance types, unsupported regions, malformed Docker images) need validation before provisioning.

7. Security Group Management Complexity (Medium Priority)

The spec says "SSH inbound from caller IP only" — this requires creating or updating security groups per-environment and handling IP changes (user moves to different network during an active session).

8. Remote Event Types Have No Handlers (Low Priority — Expected)

The 4 remote event types are defined in the EventType union but have no registered handlers. The consumer will log a warning for unhandled types. This is expected — handlers will be added in Phase 5.

9. No User-Data Script Template (Medium Priority)

The provisioning flow spec is detailed but there's no actual user-data script template. This is a complex bash script that needs to handle: Docker installation, image pulling, container startup, repo cloning, dependency installation, fuel-code/Claude Code installation, config sync, health checking, and error reporting.

10. No Cleanup/Garbage Collection (Low Priority)

No mechanism for cleaning up orphaned resources: terminated EC2 instances not properly cleaned up, S3 SSH keys from terminated environments, security groups from long-destroyed environments.

---

17. References

File	What It Contains
`tasks/CORE.md` lines 116-154	Blueprint abstraction definition, env.yaml format
`tasks/CORE.md` lines 523-563	`remote_envs` and `blueprints` table schemas
`tasks/CORE.md` lines 568-596	S3 storage layout (ssh-keys, manifests)
`tasks/CORE.md` lines 797-848	Remote provisioning flow (4 stages)
`tasks/CORE.md` lines 841-848	Remote device symmetry principle
`tasks/CORE.md` lines 852-930	CLI commands including blueprint and remote
`packages/shared/src/types/event.ts`	4 remote.* event type definitions

`packages/server/src/db/migrations/001_initial.sql`	sessions.remote_env_id placeholder, device type/status constraints
`packages/server/src/ws/websocket.ts`	broadcastRemoteUpdate() forward declaration
`tasks/phase-5/`	15 task files for Phase 5 implementation plan