fuel-code: Full-Stack Architecture Reference

> ***Version**: As of commit `6cfc158` (2026-02-27) **Audience**: Anyone who needs to make confident architectural decisions in this codebase. **Contract**: Everything stated here is backed by an exact file path and line reference. If a reference is wrong, the claim should be treated as suspect.*

Table of Contents

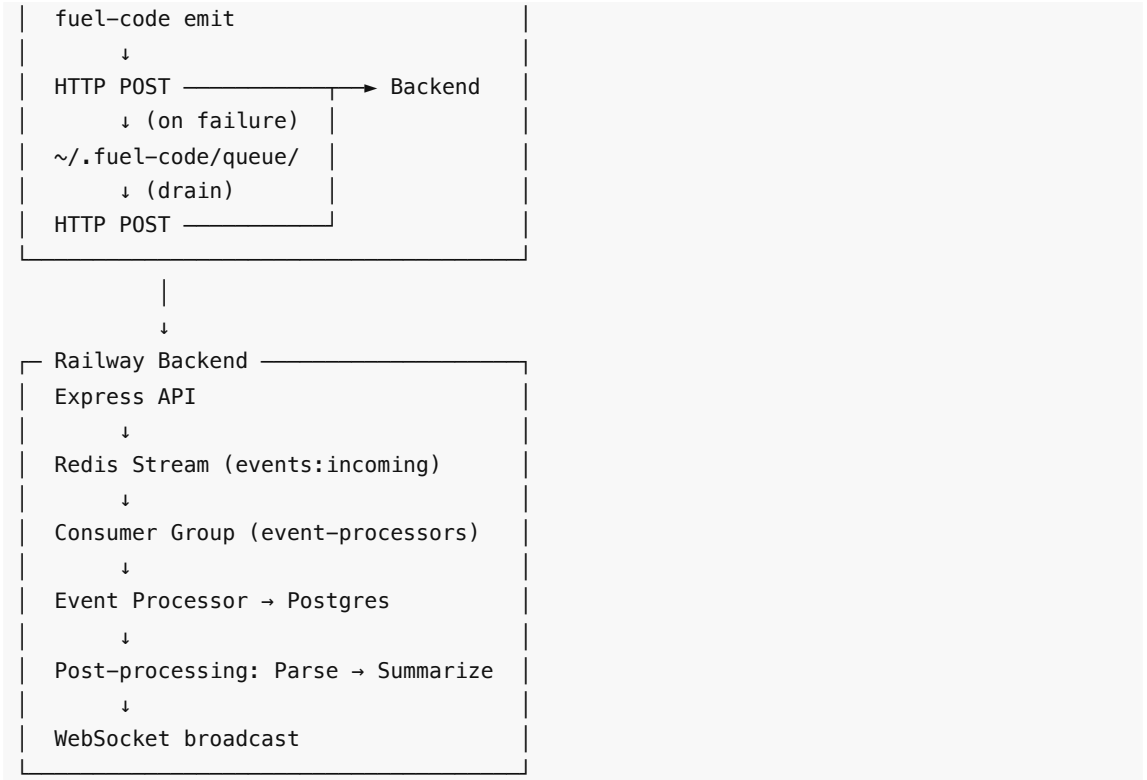
---

1. System Overview

fuel-code is a **CLI-first developer activity tracking system** that captures, stores, and surfaces Claude Code sessions as the primary unit of work. Git activity is tracked alongside sessions for full context.

Core value proposition: Every Claude Code session you run — anywhere, on any machine — is captured, parsed, summarized, and queryable.

Architecture style: Event-driven pipeline with CLI-first UI. Events flow from local hooks through an HTTP API into a Redis Stream, are processed into Postgres, and surfaced through a REST API consumed by a terminal UI.



Definitive spec: `/tasks/CORE.md` — all abstractions, schemas, flows, and invariants are documented there. This document describes the *implementation* of that spec.

---

2. Monorepo Structure & Dependency Graph

Runtime: Bun (not Node). TypeScript ESM modules throughout. **Workspace config**: Bun workspaces via root `package.json`.

```
fuel-code/
├── package.json              # workspaces: ["packages/*"]
├── tsconfig.base.json        # ESNext, strict, bundler moduleResolution
├── docker-compose.test.yml   # Postgres 16, Redis 7, LocalStack (S3)
├── tasks/CORE.md             # Definitive system spec
├── packages/
│   ├── shared/               # Types, schemas, utilities — ZERO side effects
│   ├── core/                 # Domain logic — ZERO HTTP/UI/infrastructure knowledge
│   ├── server/               # Express + Redis consumer + WebSocket
│   ├── cli/                  # Commands + TUI (Ink/React) + local queue
│   └── hooks/                # Shell scripts for CC and git hooks
└── docs/plans/               # Implementation plans for specific changes
```

Dependency direction (strict — no cycles):

```
shared ← core ← server
shared ←──────── cli
shared ←──────── hooks (via fuel-code emit)
```

`core` depends on `shared` for types/schemas and on `postgres` for SQL, but has **zero knowledge** of Express, Redis, S3 clients, or CLI. All infrastructure is injected.

`cli` depends on `shared` for types but does **not** import from `core` or `server`. It talks to the server exclusively through the HTTP API.

Source of truth for the dependency graph — `package.json` files:

```
// packages/shared/package.json — deps: ulidx, zod (no internal deps)
// packages/core/package.json   — deps: @fuel-code/shared, @anthropic-ai/sdk,
postgres, pino
// packages/server/package.json — deps: @fuel-code/core, @fuel-code/shared, express,
ioredis, ws, ...
// packages/cli/package.json    — deps: @fuel-code/shared, commander, ink, react,
ws, ...
```

(`packages/server/package.json:12–27` , `packages/cli/package.json:12–24` , `packages/core/package.json:12–16` , `packages/shared/package.json:12–14`)

TypeScript config (`tsconfig.base.json`):

```
{
  "compilerOptions": {
    "target": "ESNext",
    "module": "ESNext",
    "moduleResolution": "bundler",
    "strict": true,
    "paths": {
      "@fuel-code/shared": ["./packages/shared/src"],
      "@fuel-code/core": ["./packages/core/src"]
    }
  }
}
```

(`tsconfig.base.json:1–23`)

---

3. The Five Abstractions

Everything in the system is built from five concepts, as defined in `tasks/CORE.md:11–177` :

Abstraction	Identity	Purpose
Workspace	`canonical_id` from normalized git remote URL	The organizing principle — a project/repo
Session	Claude Code's `session_id` (UUID)	A single Claude Code invocation, the primary unit of work
Event	ULID (client-generated)	Immutable, append-only activity records

Device	ULID (generated at `fuel-code init`)	A machine that reports events (local laptop or remote EC2)
Blueprint	Auto-detected from repo	Environment config for remote dev envs (Phase 5, not yet implemented)

Workspace identity is deterministic and computed client-side:

```
// packages/shared/src/canonical.ts:113-140
export function deriveWorkspaceCanonicalId(
  remoteUrl: string | null,
  firstCommitHash: string | null,
): string {
  // Priority 1: normalize remote URL → "github.com/user/repo"
  // Priority 2: local repo → "local:<sha256-of-first-commit>"
  // Priority 3: "_unassociated"
}
```

Normalization handles SSH, HTTPS, git://, and SCP-style URLs; lowercases the host; strips `.git` suffix:

```
// packages/shared/src/canonical.ts:36-111
export function normalizeGitRemote(url: string): string | null {
  // git@github.com:user/repo.git     → github.com/user/repo
  // https://github.com/user/repo.git → github.com/user/repo
  // ssh://git@github.com/user/repo   → github.com/user/repo
}
```

---

4. Package: `shared` — Contract Layer

Location: `packages/shared/src/` **Dependencies**: `ulidx`, `zod` — no internal dependencies, zero side effects.

Types (`types/`)

File	Exports
`types/event.ts`	`EventType` (14 values), `Event` interface
`types/session.ts`	`SessionLifecycle` type, `Session` interface
`types/workspace.ts`	`Workspace` interface, `UNASSOCIATED_WORKSPACE = "_unassociated"`
`types/device.ts`	`Device` interface, `DeviceType`, `DeviceStatus`
`types/transcript.ts`	`TranscriptMessage`, `ParsedContentBlock`, `ParseResult`, `TranscriptStats`

Event types (`packages/shared/src/types/event.ts`):

```
session.start     session.end      session.compact
git.commit        git.push         git.checkout       git.merge
remote.provision.start  remote.provision.ready  remote.provision.error
```

```
remote.terminate
system.device.register  system.hooks.installed  system.heartbeat
```

Schemas (`schemas/`)

Event envelope (`packages/shared/src/schemas/event-base.ts:36-61`):

```ts
export const eventSchema = z.object({
  id: z.string().min(1),
  type: z.string().min(1),
  timestamp: z.string().datetime(),
  device_id: z.string().min(1),
  workspace_id: z.string().min(1),
  session_id: z.string().nullable(),
  data: z.record(z.unknown()),
  blob_refs: z.array(z.unknown()).default([]),
});

export const ingestRequestSchema = z.object({
  events: z.array(eventSchema).min(1).max(100),
});
```

Per-type payload validation happens separately via `validateEventPayload(type, data)` .

Utilities

File	Function	Purpose
`canonical.ts`	`normalizeGitRemote()`, `deriveWorkspaceCanonicalId()`, `deriveDisplayName()`	Workspace identity
`ulid.ts`	`generateId()`, `isValidUlid()`, `extractTimestamp()`	ULID generation via `ulidx`
`s3-keys.ts`	`buildTranscriptKey()`, `buildParsedBackupKey()`, `buildArtifactKey()`	S3 key patterns
`errors.ts`	`FuelCodeError` hierarchy	Structured error classes

Error Hierarchy

```ts
// packages/shared/src/errors.ts
FuelCodeError (base: code, message, context, statusCode)
  ├── ConfigError    (code prefix: CONFIG_*)    → HTTP 500
  ├── NetworkError   (code prefix: NETWORK_*)   → HTTP 502
  ├── ValidationError(code prefix: VALIDATION_*)→ HTTP 400
  └── StorageError   (code prefix: STORAGE_*)   → HTTP 503
```

---

5. Package: `core` — Domain Logic

Location: `packages/core/src/` **Dependencies**: `@fuel-code/shared`, `@anthropic-ai/sdk`, `postgres`, `pino` **Key constraint**: **Zero knowledge of HTTP, WebSocket, CLI, TUI, Redis, or S3 client implementations.** All infrastructure is injected via interfaces.

Event Processor

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

The central dispatch function called by the Redis consumer for every event:

```
// event-processor.ts:132-235
export async function processEvent(
  sql: Sql,
  event: Event,
  registry: EventHandlerRegistry,
  logger: Logger,
  pipelineDeps?: PipelineDeps,
): Promise<ProcessResult>
```

Steps:

1. `resolveOrCreateWorkspace()` — canonical string → ULID upsert
2. `resolveOrCreateDevice()` — ID → device row upsert
3. `ensureWorkspaceDeviceLink()` — junction table upsert
4. Strip `_device_name` / `_device_type` transport hints from `event.data`
5. `INSERT INTO events ... ON CONFLICT (id) DO NOTHING` — ULID-based dedup
6. `validateEventPayload()` — type-specific Zod check
7. Dispatch to `registry.getHandler(event.type)` — extensible handler pattern

Handler Registry (`packages/core/src/handlers/index.ts:31-39`):

```
registry.register("session.start", handleSessionStart);
registry.register("session.end", handleSessionEnd);
registry.register("git.commit", handleGitCommit);
registry.register("git.push", handleGitPush);
registry.register("git.checkout", handleGitCheckout);
registry.register("git.merge", handleGitMerge);
```

Unknown event types pass through silently (no handler = no-op after persistence).

Workspace Resolver

File: `packages/core/src/workspace-resolver.ts`

```
-- workspace-resolver.ts:52
INSERT INTO workspaces (id, canonical_id, display_name, ...)
VALUES ($1, $2, $3, ...)
ON CONFLICT (canonical_id) DO UPDATE SET updated_at = now()
RETURNING id
```

Upsert by `canonical_id` — safe under concurrency. Returns the ULID.

Device Resolver

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

```
-- device-resolver.ts:49
INSERT INTO devices (id, name, type, hostname, os, arch, ...)
ON CONFLICT (id) DO UPDATE SET
  name = COALESCE(EXCLUDED.name, devices.name),
  hostname = COALESCE(EXCLUDED.hostname, devices.hostname),
  ...
```

`COALESCE(EXCLUDED.x, devices.x)` — fills NULL fields without overwriting existing values.

Session Handlers

`session.start` handler (`packages/core/src/handlers/session-start.ts`):

- Inserts session row with `lifecycle='detected'` , `ON CONFLICT (id) DO NOTHING` for idempotency
- Calls `checkGitHooksPrompt()` to flag workspace for git hook installation prompt

`session.end` handler (`packages/core/src/handlers/session-end.ts`):

- `transitionSession(['detected', 'capturing'] → 'ended')` with optimistic locking
- If `pipelineDeps` present and session has `transcript_s3_key` : triggers post-processing pipeline (backfill path)

Git Handlers

`git.commit` handler (`packages/core/src/handlers/git-commit.ts`):

- `correlateGitEventToSession()` — links commit to active CC session (heuristic)
- Transaction: INSERT `git_activity` , UPDATE `events.session_id`

Git correlation (`packages/core/src/git-correlator.ts:53-67`):

```
SELECT id FROM sessions
WHERE workspace_id = $1 AND device_id = $2
  AND lifecycle IN ('detected', 'capturing')
  AND started_at <= $3
ORDER BY started_at DESC LIMIT 1
```

Heuristic: most recent active session on same device+workspace. Works because git operations and CC sessions co-occur on the same device.

Transcript Parser

File: `packages/core/src/transcript-parser.ts` **Design**: Pure function — zero I/O, zero DB, no side effects.

```
// transcript-parser.ts:73-77
export async function parseTranscript(
  sessionId: string,
```

```
  input: string | ReadableStream,
  options?: ParseOptions,
): Promise<ParseResult>
```

Three-pass algorithm:

1. **Pass 1** (lines 124-197): JSON-parse each JSONL line, skip internal types (`progress` , `file-history-snapshot` , `queue-operation`), group assistant lines by `message.id` (CC streams multi-line assistant responses sharing the same `message.id`)
2. **Pass 2** (lines 204-247): Build `TranscriptMessage` and `ParsedContentBlock` rows. For assistant groups, merges content blocks from all streaming lines, takes token usage from the LAST line (most complete)
3. **Pass 3** (line 253): `computeStats()` — aggregates totals (messages, tool uses, thinking blocks, tokens, cost, duration)

Content block types: `text` , `thinking` , `tool_use` , `tool_result` **Tool result truncation**: Results > 256 KB are truncated, flagged with `metadata.truncated = true`

Cost pricing (transcript-parser.ts:38-41):

```
const PRICE_INPUT_PER_MTOK = 3.0;
const PRICE_OUTPUT_PER_MTOK = 15.0;
const PRICE_CACHE_READ_PER_MTOK = 0.3;
const PRICE_CACHE_WRITE_PER_MTOK = 3.75;
```

Summary Generator

File: `packages/core/src/summary-generator.ts` **Design**: Pure function, never throws, always returns `SummaryResult` .

```
// summary-generator.ts:50
const SUMMARY_SYSTEM_PROMPT = `You are a technical activity summarizer. Write a
1-3 sentence summary of this Claude Code session in past tense. Focus on WHAT was
accomplished, not HOW. Be specific about files, features, or bugs. Do not start
with "The user" or "This session".`;
```

Rendering rules for the condensed prompt:

- User messages: first 500 chars
- Assistant text blocks: first 300 chars
- Tool use: `"- Used {tool_name}"` (name only)
- Thinking blocks and tool results: skipped entirely
- Max 8000 chars total; if exceeded, keeps first 3000 + last 3000 with truncation marker

API call: `claude-sonnet-4-5-20250929` , `temperature: 0.3` , `max_tokens: 150` , 30s timeout via `AbortSignal.timeout()` .

Session Recovery

File: `packages/core/src/session-recovery.ts`

On server startup (5s delay after boot):

- `recoverStuckSessions()` : finds sessions stuck in `ended` / `parsed` with stale `parse_status` , re-enqueues them
- `recoverUnsummarizedSessions()` : finds `lifecycle='parsed'` with `summary IS NULL` , resets and re-enqueues

---

6. Package: `server` — HTTP + Redis + WebSocket

Location: `packages/server/src/` **Runtime**: Bun + Express 5 (not Express 4) **Dependencies**: `@fuel-code/core` , `@fuel-code/shared` , Express, ioredis, ws, AWS SDK v3, postgres.js, pino, helmet, cors, zod

Server Entry Point

File: `packages/server/src/index.ts`

Ordered startup sequence (index.ts:78-219):

```
Step 1-2: validateEnv() — requires DATABASE_URL, REDIS_URL, API_KEY, PORT
Step 3:   createDb(DATABASE_URL) — postgres.js pool, max 10 connections
Step 4:   runMigrations(sql, migrationsDir) — aborts on error
Step 5:   createRedisClient × 2 — one for HTTP ops, one blocking for consumer
Step 6:   ensureConsumerGroup(redis) — XGROUP CREATE with MKSTREAM
Step 7:   loadS3Config() + createS3Client() + s3.ensureBucket()
Step 7b:  createPipelineQueue(3) — bounded queue, 3 concurrent, 50 max pending
Step 8:   createApp(...) + createServer(app) + createWsServer(...)
Step 9:   startConsumer(...) — Redis Stream consumer loop
Step 11:  (5s delay) recoverStuckSessions + recoverUnsummarizedSessions
```

Two Redis clients (index.ts:115-118): The consumer uses `XREADGROUP BLOCK` which holds the connection, so health checks and event writes need a separate non-blocking connection.

Graceful shutdown (index.ts:224-271): On SIGTERM/SIGINT:

1. `httpServer.close()` — stop new HTTP connections
2. `wsServer.shutdown()` — close all WS clients, clear ping interval
3. `pipelineQueue.stop()` — drain in-flight pipelines (no new work)
4. `consumer.stop()` — wait for current iteration (10s timeout)
5. `redis.disconnect()` × 2
6. `sql.end()` — close Postgres pool
7. Force exit after 30s if cleanup hangs

Express App & Middleware

File: `packages/server/src/app.ts`

Middleware stack (in order, app.ts:65-141):

```
1. express.json({ limit: "1mb" })     — body parsing, 1MB cap
2. helmet()                            — security headers (HSTS, X-Frame-Options,
etc.)
3. cors({ origin: false })             — CORS disabled (no web client yet)
4. pinoHttp({ ... })                   — request logging, skips /api/health
5. GET /api/health                     — BEFORE auth (unauthenticated, for Railway
probes)
```

```
6. Bearer auth middleware              — on all remaining /api/* routes
7. Routes (events, sessions, transcript-upload, session-actions, timeline, workspaces,
devices, prompts)
8. errorHandler                        — MUST be last
```

All API Endpoints

Method	Path	Purpose	File
GET	/api/health	Health check (unauthenticated)	routes/health.ts
POST	/api/events/ingest	Batch event ingestion → Redis Stream	routes/events.ts
GET	/api/sessions	List sessions, cursor-based pagination	routes/sessions.ts
GET	/api/sessions/:id	Session detail with workspace+device names	routes/sessions.ts
GET	/api/sessions/:id/transcript	Parsed messages + content blocks (JSON)	routes/sessions.ts
GET	/api/sessions/:id/transcript/raw	Presigned S3 URL for raw JSONL	routes/sessions.ts
GET	/api/sessions/:id/events	Raw events for a session	routes/sessions.ts
GET	/api/sessions/:id/git	Git activity for a session	routes/sessions.ts
PATCH	/api/sessions/:id	Update tags or summary	routes/sessions.ts
POST	/api/sessions/:id/transcript/upload	Upload raw JSONL transcript	routes/transcript-upload.ts
POST	/api/sessions/:id/reparse	Re-trigger transcript parsing	routes/session-actions.ts
GET	/api/timeline	Unified activity feed (sessions + orphan git)	routes/timeline.ts
GET	/api/workspaces	List workspaces with aggregate stats	routes/workspaces.ts
GET	/api/workspaces/:id	Workspace detail: sessions, devices, git, stats	routes/workspaces.ts
GET	/api/devices	List devices with CTE-based aggregate counts	routes/devices.ts

GET	/api/devices/:id	Device detail: workspaces and recent sessions	routes/devices.ts
GET	/api/prompts/pending	Pending git hook install prompts	routes/prompts.ts
POST	/api/prompts/dismiss	Accept or decline a prompt	routes/prompts.ts
WS	/api/ws	WebSocket for real-time updates	ws/index.ts

Cursor-Based Pagination

Used by sessions, timeline, and workspaces endpoints:

```
// routes/sessions.ts:68-89
// Cursor = base64({ s: started_at, i: id })
// Keyset pagination on (started_at DESC, id DESC)
```

Transcript Upload Endpoint

File: `packages/server/src/routes/transcript-upload.ts`

```
POST /api/sessions/:id/transcript/upload
1. Validate Content-Length present and <= 200MB
2. Verify session exists
3. Idempotent: if transcript_s3_key already set, return 200
4. Buffer request body in memory chunks, upload to S3
5. S3 key: transcripts/{canonicalId}/{sessionId}/raw.jsonl
6. Update sessions.transcript_s3_key
7. If session lifecycle is "ended": trigger pipeline via bounded queue
8. Return 202
```

Health Check

File: `packages/server/src/routes/health.ts`

Runs `checkDbHealth()` and `checkRedisHealth()` in parallel:

- Both OK → `"ok"` (200)
- Redis down, DB OK → `"degraded"` (200)
- DB down → `"unhealthy"` (503)

Response includes `ws_clients`, `uptime_seconds`, `version`.

Error Handler

File: `packages/server/src/middleware/error-handler.ts`

Error Type	HTTP Status

`ZodError`	400 with `details: err.issues`
`ValidationError` (VALIDATION_*)	400
`ConfigError` (CONFIG_*)	500
`NetworkError` (NETWORK_*)	502
`StorageError` (STORAGE_*)	503
Unknown	500

Stack traces included only in non-production.

---

7. Package: `cli` — Commands, TUI, Hooks

Location: `packages/cli/src/` **Entry point**: `packages/cli/src/index.ts` (shebang: `#!/usr/bin/env bun`) **CLI framework**: Commander

Command Registry

Command	File	Purpose
`(no args)`	`tui/App.tsx`	Launch Ink TUI dashboard
`init`	`commands/init.ts`	Initialize device, write `~/.fuel-code/config.yaml`
`status`	`commands/status.ts`	Device info, backend health, sessions, queue
`sessions`	`commands/sessions.ts`	List/filter sessions
`session <id>`	`commands/session-detail.ts`	Session detail with flags
`timeline`	`commands/timeline.ts`	Unified timeline view
`workspaces` / `workspace`	`commands/workspaces.ts`	Workspace list + detail
`emit`	`commands/emit.ts`	Emit an event (used by hooks internally)
`queue`	`commands/queue.ts`	Queue status/drain/dead-letter management
`hooks`	`commands/hooks.ts`	Install/uninstall/status/test CC + git hooks
`cc-hook`	`commands/cc-hook.ts`	Internal handler called by Claude Code hooks
`transcript`	`commands/transcript.ts`	Upload JSONL transcript files
`backfill`	`commands/backfill.ts`	Historical session discovery and ingestion

Configuration

File: `packages/cli/src/lib/config.ts` **Location**: `~/.fuel-code/config.yaml`

```yaml
backend:
  url: "https://..."        # backend origin
  api_key: "fc_..."         # Bearer token
```

```yaml
device:
  id: "01HZ..."             # ULID, generated once at init
  name: "macbook-pro"       # hostname by default
  type: "local"             # or "remote"
pipeline:
  queue_path: "~/.fuel-code/queue"
  drain_interval_seconds: 10
  batch_size: 50
  post_timeout_ms: 5000
```

Security: `saveConfig()` writes atomically (tmp + rename) and sets permissions to `0o600` (owner read/write only).

Event Emission Path

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

This is the **single ingestion pathway** for all hook-generated events:

```
1. loadConfig() → FuelCodeConfig
2. Parse --data JSON (or --data-stdin for git hooks)
3. Inject _device_name + _device_type hints into payload
4. Build Event object { id: ULID, type, timestamp, device_id, workspace_id, ... }
5. POST to backend via createApiClient(config).ingest([event])
   - Timeout: config.pipeline.post_timeout_ms (default 5000ms)
6. On failure → enqueueEvent(event, queueDir)   // disk fallback
7. Always exit 0
```

Key constraint: The command is designed to **always exit 0**. Any failure falls through to local queue. This is critical because hooks must never block Claude Code or git operations.

Pre-Action Prompt Hook

```typescript
// packages/cli/src/index.ts:139-158
program.hook("preAction", async (thisCommand) => {
  // Only for interactive commands (sessions, status, hooks, backfill, etc.)
  // NOT for fire-and-forget commands (emit, transcript, queue)
  const prompts = await checkPendingPrompts(config);
  for (const prompt of prompts) {
    if (prompt.type === "git_hooks_install") {
      await showGitHooksPrompt(prompt, config);
    }
  }
});
```

---

8. Package: `hooks` — Claude Code & Git Scripts

Claude Code Hooks

Installation target: `~/.claude/settings.json` under `hooks.SessionStart` and `hooks.SessionEnd`.

How CC hooks are installed (`packages/cli/src/commands/hooks.ts:195-196`):

```
bash -c 'data=$(cat); printf "%s" "$data" | ${cliCommand} cc-hook session-start &'
bash -c 'data=$(cat); printf "%s" "$data" | ${cliCommand} cc-hook session-end &'
```

Critical design: `data=$(cat)` captures stdin synchronously before backgrounding (`&`), because Claude Code pipes context JSON to the hook's stdin.

CC hook handler (`packages/cli/src/commands/cc-hook.ts`):

On `session-start` :

1. Read stdin JSON from CC (contains `session_id` , `cwd` , `transcript_path` , `source` , `model`)
2. `resolveWorkspace(cwd)` — runs `git remote` , `git symbolic-ref` , etc.
3. Get CC version via `claude --version`
4. Call `runEmit("session.start", {...})`

On `session-end` :

1. Read stdin JSON
2. Call `runEmit("session.end", {...})` with `sessionId`
3. Call `runTranscriptUpload(sessionId, transcriptPath)` with 120s timeout

Git Hooks

Location: `packages/hooks/git/` **Scripts**: `post-commit` , `post-checkout` , `post-merge` , `pre-push` , `resolve-workspace.sh`

Installation (`packages/cli/src/lib/git-hook-installer.ts`):

Global mode (default):

1. Create `~/.fuel-code/git-hooks/` directory
2. Detect competing hook managers (Husky, Lefthook, pre-commit) — abort unless `--force`
3. Back up existing hooks as `<hook>.user` for chaining
4. Copy scripts + chmod 755
5. `git config --global core.hooksPath ~/.fuel-code/git-hooks`

Hook chaining: Each hook checks for `<hook>.user` and runs it first, preserving existing hooks.

All git hooks:

- Check `fuel-code` in PATH, check per-repo opt-out (`.fuel-code/config.yaml` with `git_enabled: false`)
- Resolve workspace via `resolve-workspace.sh`
- Fire `fuel-code emit` in a background subshell (`&`)
- Redirect output to `~/.fuel-code/hook-errors.log`
- Always exit 0 (except `pre-push` which passes through user hook exit codes)

Workspace resolution in shell (`packages/hooks/git/resolve-workspace.sh`):

```
# 1. git remote get-url origin (or first remote alphabetically)
# 2. Normalize SSH/HTTPS → host/user/repo
```

```
# 3. If no remote: "local:<sha256 of first commit hash>"
# 4. If no commits: exit 1
```

---

9. Data Flow: End-to-End Event Pipeline

Here is the complete path of an event from hook fire to queryable data:

```
┌─ LOCAL MACHINE ──────────────────────────────────────────────────┐
│                                                                  │
│  1. TRIGGER                                                      │
│     CC SessionStart hook fires (or git post-commit, etc.)        │
│     → stdin JSON: { session_id, cwd, transcript_path, ... }      │
│                                                                  │
│  2. CC-HOOK HANDLER (packages/cli/src/commands/cc-hook.ts)       │
│     → reads stdin, resolves workspace (git remote → canonical ID)│
│     → calls runEmit("session.start", payload)                    │
│                                                                  │
│  3. EMIT (packages/cli/src/commands/emit.ts)                     │
│     → builds Event { id: ULID, type, timestamp, device_id, ... } │
│     → POST /api/events/ingest with Bearer token                  │
│     → on failure: enqueueEvent() to ~/.fuel-code/queue/          │
│     → always exit 0                                              │
│                                                                  │
│  4. QUEUE DRAIN (if events were queued)                          │
│     → spawnBackgroundDrain() — detached bun child process        │
│     → reads queue files in ULID order, POSTs in batches of 50    │
│     → on success: delete files. After 100 fails: dead-letter.    │
└──────────────────────────────────────────────────────────────────┘
          │
          │  HTTP POST /api/events/ingest
          ↓
┌─ RAILWAY BACKEND ────────────────────────────────────────────────┐
│                                                                  │
│  5. INGESTION ROUTE (packages/server/src/routes/events.ts)       │
│     → Zod validate request envelope (ingestRequestSchema)       │
│     → Per-event payload validation (validateEventPayload)       │
│     → publishBatchToStream() — Redis pipeline XADD               │
│     → Return HTTP 202 with per-event accept/reject results      │
│                                                                  │
│  6. REDIS STREAM (events:incoming)                               │
│     → Durable, ordered stream of event entries                   │
│     → Consumer group: "event-processors"                         │
│                                                                  │
│  7. CONSUMER LOOP (packages/server/src/pipeline/consumer.ts)     │
│     → XREADGROUP BLOCK 5000 — reads up to 10 entries             │
│     → For each entry: handleEntry()                              │
│       → processEvent() from core                                 │
│       → On success: XACK + broadcast via WebSocket               │
│       → On failure: retry (3 max, then dead-letter via XACK)     │
│                                                                  │
│  8. EVENT PROCESSOR (packages/core/src/event-processor.ts)       │
```

```
│     → Resolve/create workspace (canonical_id → ULID upsert)          │
│     → Resolve/create device (upsert)                                  │
│     → Link workspace ↔ device                                         │
│     → INSERT INTO events ON CONFLICT DO NOTHING (dedup)               │
│     → Dispatch to type-specific handler                               │
│                                                                       │
│  9. TYPE HANDLER (packages/core/src/handlers/*)                       │
│     session.start → INSERT session row (lifecycle=detected)           │
│     session.end   → transition lifecycle to 'ended'                   │
│     git.commit    → INSERT git_activity, correlate to session         │
│                                                                       │
│  10. POST-PROCESSING (triggered by session.end or transcript upload)  │
│      → Download JSONL from S3                                         │
│      → parseTranscript() → messages + content blocks                  │
│      → Batch INSERT to transcript_messages + content_blocks           │
│      → Transition: ended → parsed                                     │
│      → generateSummary() via Claude Sonnet API                        │
│      → Transition: parsed → summarized                                │
│                                                                       │
│  11. WEBSOCKET BROADCAST                                              │
│      → broadcastEvent(event) to subscribed clients                    │
│      → broadcastSessionUpdate(id, lifecycle) on state changes         │
└───────────────────────────────────────────────────────────────────────┘
```

---

10. Database Schema (Postgres)

All migrations in `packages/server/src/db/migrations/`.

Migration System

File: `packages/server/src/db/migrator.ts`

- Uses `pg_advisory_lock(48756301)` to prevent concurrent migration runs
- Reads `.sql` files lexicographically from the migrations directory
- Each migration runs in its own transaction
- Tracks applied migrations in `_migrations` table

Table: `workspaces` (001_initial.sql)

```sql
CREATE TABLE workspaces (
    id              TEXT PRIMARY KEY,        -- ULID
    canonical_id    TEXT NOT NULL UNIQUE,     -- normalized git remote URL
    display_name    TEXT NOT NULL,            -- derived from repo name
    default_branch  TEXT,
    metadata        JSONB NOT NULL DEFAULT '{}',
    first_seen_at   TIMESTAMPTZ NOT NULL DEFAULT now(),
    updated_at      TIMESTAMPTZ NOT NULL DEFAULT now()
);
```

Table: `devices` (001_initial.sql)

```
CREATE TABLE devices (
    id              TEXT PRIMARY KEY,        -- ULID, generated client-side
    name            TEXT NOT NULL,
    type            TEXT NOT NULL CHECK (type IN ('local', 'remote')),
    hostname        TEXT,
    os              TEXT,                    -- "darwin", "linux"
    arch            TEXT,                    -- "arm64", "x86_64"
    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()
);
```

Table: `workspace_devices` (001_initial.sql + 004)

```
CREATE TABLE workspace_devices (
    workspace_id    TEXT NOT NULL REFERENCES workspaces(id),
    device_id       TEXT NOT NULL REFERENCES devices(id),
    local_path      TEXT NOT NULL,
    hooks_installed BOOLEAN NOT NULL DEFAULT false,
    git_hooks_installed BOOLEAN NOT NULL DEFAULT false,
    last_active_at  TIMESTAMPTZ NOT NULL DEFAULT now(),
    -- Added in migration 004:
    pending_git_hooks_prompt BOOLEAN NOT NULL DEFAULT false,
    git_hooks_prompted       BOOLEAN NOT NULL DEFAULT false,
    PRIMARY KEY (workspace_id, device_id)
);
```

Table: `sessions` (001_initial.sql)

```
CREATE TABLE sessions (
    id              TEXT PRIMARY KEY,        -- CC's session ID (UUID text)
    workspace_id    TEXT NOT NULL REFERENCES workspaces(id),
    device_id       TEXT NOT NULL REFERENCES devices(id),
    remote_env_id   TEXT,                    -- no FK yet (Phase 5)

    -- Lifecycle state machine
    lifecycle       TEXT NOT NULL DEFAULT 'detected'
                    CHECK (lifecycle IN (
                        'detected', 'capturing', 'ended',
                        'parsed', 'summarized', 'archived', 'failed'
                    )),
    started_at      TIMESTAMPTZ NOT NULL,
    ended_at        TIMESTAMPTZ,
    end_reason      TEXT,

    -- Context
```

```
    initial_prompt  TEXT,
    git_branch      TEXT,
    model           TEXT,
    source          TEXT,

    -- Transcript storage
    transcript_s3_key TEXT,
    parse_status    TEXT DEFAULT 'pending'
                    CHECK (parse_status IN ('pending', 'parsing', 'completed',
'failed')),
    parse_error     TEXT,
    summary         TEXT,

    -- Aggregate stats (populated after parsing)
    total_messages      INTEGER,
    user_messages       INTEGER,
    assistant_messages  INTEGER,
    tool_use_count      INTEGER,
    thinking_blocks     INTEGER,
    subagent_count      INTEGER,
    tokens_in           BIGINT,
    tokens_out          BIGINT,
    cache_read_tokens   BIGINT,
    cache_write_tokens  BIGINT,
    cost_estimate_usd   NUMERIC(10, 6),
    duration_ms         INTEGER,

    tags            TEXT[] NOT NULL DEFAULT '{}',
    metadata        JSONB NOT NULL DEFAULT '{}',
    created_at      TIMESTAMPTZ NOT NULL DEFAULT now(),
    updated_at      TIMESTAMPTZ NOT NULL DEFAULT now()
);
```

Indexes:

- `idx_sessions_workspace` — `(workspace_id, started_at DESC)`
- `idx_sessions_device` — `(device_id, started_at DESC)`
- `idx_sessions_lifecycle` — `(lifecycle)`
- `idx_sessions_tags` — GIN on `tags` array for `@>` containment queries

Table: `events` (001_initial.sql)

```
CREATE TABLE events (
    id              TEXT PRIMARY KEY,        -- ULID, client-generated
    type            TEXT NOT NULL,
    timestamp       TIMESTAMPTZ NOT NULL,    -- when it happened on source machine
    device_id       TEXT NOT NULL REFERENCES devices(id),
    workspace_id    TEXT NOT NULL REFERENCES workspaces(id),
    session_id      TEXT REFERENCES sessions(id),  -- nullable for non-session
events
    data            JSONB NOT NULL,
```

```
    blob_refs       JSONB NOT NULL DEFAULT '[]',
    ingested_at     TIMESTAMPTZ NOT NULL DEFAULT now()
);
```

Indexes:

- `idx_events_workspace_time` — `(workspace_id, timestamp DESC)`
- `idx_events_session` — `(session_id, timestamp ASC) WHERE session_id IS NOT NULL`
- `idx_events_type` — `(type, timestamp DESC)`
- `idx_events_device` — `(device_id, timestamp DESC)`

Table: `transcript_messages` (002_transcript_tables.sql)

```
CREATE TABLE transcript_messages (
    id              TEXT PRIMARY KEY,        -- ULID
    session_id      TEXT NOT NULL REFERENCES sessions(id) ON DELETE CASCADE,
    line_number     INTEGER NOT NULL,
    ordinal         INTEGER NOT NULL,        -- conversation order
    message_type    TEXT NOT NULL,           -- "user" | "assistant" | "system" |
"summary"
    role            TEXT,
    model           TEXT,
    tokens_in       INTEGER,
    tokens_out      INTEGER,
    cache_read      INTEGER,
    cache_write     INTEGER,
    cost_usd        NUMERIC(10, 6),
    compact_sequence INTEGER NOT NULL DEFAULT 0,
    is_compacted    BOOLEAN NOT NULL DEFAULT false,
    timestamp       TIMESTAMPTZ,
    raw_message     JSONB,                   -- lossless reconstruction
    metadata        JSONB NOT NULL DEFAULT '{}',
    has_text        BOOLEAN NOT NULL DEFAULT false,
    has_thinking    BOOLEAN NOT NULL DEFAULT false,
    has_tool_use    BOOLEAN NOT NULL DEFAULT false,
    has_tool_result BOOLEAN NOT NULL DEFAULT false
);
```

Table: `content_blocks` (002_transcript_tables.sql)

```
CREATE TABLE content_blocks (
    id              TEXT PRIMARY KEY,        -- ULID
    message_id      TEXT NOT NULL REFERENCES transcript_messages(id) ON DELETE
CASCADE,
    session_id      TEXT NOT NULL REFERENCES sessions(id) ON DELETE CASCADE,
    block_order     INTEGER NOT NULL,
    block_type      TEXT NOT NULL,           -- "text" | "thinking" | "tool_use" |
"tool_result"
    content_text    TEXT,
    thinking_text   TEXT,
```

```
    tool_name       TEXT,
    tool_use_id     TEXT,
    tool_input      JSONB,
    tool_result_id  TEXT,
    is_error        BOOLEAN DEFAULT false,
    result_text     TEXT,
    result_s3_key   TEXT,                    -- S3 key if result was too large
    metadata        JSONB NOT NULL DEFAULT '{}'
);
```

Notable indexes:

- `idx_content_blocks_text` — GIN `to_tsvector('english', content_text) WHERE content_text IS NOT NULL` — full-text search on content
- `idx_content_blocks_tool` — `(tool_name) WHERE tool_name IS NOT NULL`
- `idx_sessions_needs_recovery` — `(lifecycle, updated_at) WHERE lifecycle IN ('ended','parsed') AND parse_status IN ('pending','parsing')` — for startup recovery

Table: `git_activity` (003_create_git_activity.sql)

```
CREATE TABLE git_activity (
    id              TEXT PRIMARY KEY,        -- same as the event ID
    workspace_id    TEXT NOT NULL REFERENCES workspaces(id),
    device_id       TEXT NOT NULL REFERENCES devices(id),
    session_id      TEXT REFERENCES sessions(id),  -- nullable for "orphan" events
    type            TEXT NOT NULL CHECK (type IN
('commit','push','checkout','merge')),
    branch          TEXT,
    commit_sha      TEXT,
    message         TEXT,
    files_changed   INTEGER,
    insertions      INTEGER,
    deletions       INTEGER,
    timestamp       TIMESTAMPTZ NOT NULL,
    data            JSONB NOT NULL DEFAULT '{}',
    created_at      TIMESTAMPTZ NOT NULL DEFAULT now()
);
```

Tables Specified but Not Yet Created

From `tasks/CORE.md:523–563` (Phase 5):

- `remote_envs` — disposable cloud dev boxes
- `blueprints` — saved environment configurations

These exist in the spec but have no migration files yet.

---

11. Redis Streams Architecture

File: `packages/server/src/redis/stream.ts`

Configuration Constants

```
// stream.ts:28, 31, 38
export const EVENTS_STREAM = "events:incoming";
export const CONSUMER_GROUP = "event-processors";
export const CONSUMER_NAME = `${hostname()}-${process.pid}`;
```

Publishing

`publishBatchToStream()` (stream.ts:180-224): Uses ioredis pipeline to batch multiple `XADD` commands in a single round-trip. Events serialized to flat key-value pairs; `data` and `blob_refs` JSON-encoded as strings. `null` → empty string.

Consuming

`readFromStream()` (stream.ts:263-294): `XREADGROUP GROUP event-processors ${CONSUMER_NAME} COUNT 10 BLOCK 5000 STREAMS events:incoming >` — reads up to 10 new messages, blocking up to 5 seconds.

Acknowledgement

`acknowledgeEntry()` (stream.ts:310): `XACK events:incoming event-processors ${streamId}`

Pending Entry Reclamation

`claimPendingEntries()` (stream.ts:345-451): First tries `XAUTOCLAIM` (Redis 6.2+), falls back to `XPENDING + XCLAIM`. Claims entries idle for ≥ 60 seconds. This handles crashed workers — their un-acknowledged entries are picked up by healthy workers.

Consumer Loop

File: `packages/server/src/pipeline/consumer.ts`

```
Startup:
  1. ensureConsumerGroup() — retry loop until success (5s delay between retries)
  2. claimPendingEntries(60_000, 100) — reclaim crashed worker messages

Main loop (runs until stop()):
  3. readFromStream(10, 5000) — 10 entries, 5s block
  4. For each entry → handleEntry():
     a. processEvent(sql, event, registry, logger, pipelineDeps)
     b. On success/duplicate: XACK, clear failure count
     c. On failure: increment in-memory failure count
     d. After 3 failures: XACK (dead-letter), log permanent failure
  5. On NOGROUP error: re-create consumer group, retry immediately
  6. On other errors: log + 5s wait before retry

Stats logged every 60s: processed, duplicates, errors, pending count.
```

---

12. S3 Storage Layout

File: `packages/shared/src/s3-keys.ts`

```
fuel-code-blobs/
├── transcripts/
│   └── {workspace_canonical_id}/     # e.g., "github.com/user/repo"
│       └── {session_id}/
│           ├── raw.jsonl              # Full CC transcript (source of truth, never
deleted)
│           └── parsed.json            # Backup of parsed hierarchy
├── artifacts/
│   └── {session_id}/
│       └── {artifact_id}.{ext}        # Large tool outputs (>256KB)
├── ssh-keys/                          # Phase 5 (not yet implemented)
│   └── {remote_env_id}/
└── manifests/                         # Phase 5 (not yet implemented)
    └── {workspace_canonical_id}/
```

S3 Client (`packages/server/src/aws/s3.ts`):

- AWS SDK v3 with `maxAttempts: 3` (exponential backoff)
- Supports LocalStack via `S3_ENDPOINT` + `S3_FORCE_PATH_STYLE=true`
- Methods: `upload` , `uploadStream` , `uploadFile` , `download` , `downloadStream` , `presignedUrl` , `headObject` , `delete` , `healthCheck` , `ensureBucket`

Storage rules (from CORE.md):

- Raw transcripts always go to S3, never deleted
- Parsed transcript data lives in Postgres for querying, recoverable from S3
- Tool results > 256KB go to S3 with reference in `content_blocks.result_s3_key`

---

13. Session Lifecycle State Machine

File: `packages/core/src/session-lifecycle.ts`

State Diagram

```
detected ──→ capturing ──→ ended ──→ parsed ──→ summarized ──→ archived
    │                        │         │           │
    ├──→ ended (skip)        ├──→ failed├──→ failed ├──→ failed
    └──→ failed
```

`failed` and `archived` are terminal states. Recovery from `failed` requires `resetSessionForReparse()` .

Transition Map

```ts
// session-lifecycle.ts:58-66
export const TRANSITIONS: Record<SessionLifecycle, SessionLifecycle[]> = {
  detected:   ["capturing", "ended", "failed"],
  capturing:  ["ended", "failed"],
  ended:      ["parsed", "failed"],
```

```
  parsed:     ["summarized", "failed"],
  summarized: ["archived"],
  archived:   [],
  failed:     [],
};
```

Optimistic Locking

```
// session-lifecycle.ts:127-219
// transitionSession() uses:
UPDATE sessions
SET lifecycle = $to, ...
WHERE id = $id AND lifecycle = ANY($from)
RETURNING lifecycle
```

If 0 rows returned, it queries the actual state to provide diagnostic feedback (e.g., "session was already in 'parsed'"). Concurrent calls serialize at the DB level.

Reset for Reparse

```
// session-lifecycle.ts:315
// resetSessionForReparse() runs in a transaction:
// 1. DELETE FROM content_blocks WHERE session_id = $1
// 2. DELETE FROM transcript_messages WHERE session_id = $1
// 3. UPDATE sessions SET lifecycle='ended', parse_status='pending', clear stats
```

Source states: `ended`, `parsed`, `summarized`, `failed`.

---

14. Post-Processing Pipeline (Transcript → Summary)

File: `packages/core/src/session-pipeline.ts`

Pipeline Orchestrator

`runSessionPipeline()` (session-pipeline.ts:93-323) — **never throws**:

```
Step 1: Validate session is "ended" with transcript_s3_key
Step 2: Set parse_status = 'parsing' (claim marker)
Step 3: Download transcript from S3
Step 4: parseTranscript() → messages + content blocks
Step 5: Transaction: DELETE old data, batch INSERT in chunks of 500
        → transcript_messages (21 columns per row)
        → content_blocks (14 columns per row)
Step 6: transitionSession('ended' → 'parsed') with computed stats
Step 7: generateSummary() — best-effort
        → Failure keeps session at 'parsed', does NOT regress lifecycle
Step 8: Upload parsed backup to S3 (best-effort, fire-and-forget)
```

Pipeline Queue

`createPipelineQueue(maxConcurrent: 3)` (session-pipeline.ts:469-577):

```
// Bounded: MAX_QUEUE_DEPTH = 50
// Drops entries when full (with warning log)
// Fire-and-forget: errors caught and logged
// stop() drains in-flight work before shutdown
```

Triggers

The pipeline is triggered from three places:

1. **Transcript upload route** (`routes/transcript-upload.ts`) — when session is already `"ended"`
2. **`session.end` handler** (`handlers/session-end.ts`) — when transcript already in S3 (backfill path)
3. **Session recovery on startup** (`session-recovery.ts`) — for stuck sessions

---

15. WebSocket Protocol

Files: `packages/server/src/ws/index.ts` , `ws/broadcaster.ts` , `ws/types.ts`

Connection

- Path: `/api/ws`
- Auth: `?token=<api_key>` query parameter
- Invalid token → close with code 4001
- Each client gets a ULID client ID

Client → Server Messages

```
{ type: "subscribe", scope: "all" }
{ type: "subscribe", workspace_id: "..." }
{ type: "subscribe", session_id: "..." }
{ type: "unsubscribe", workspace_id?: "...", session_id?: "..." }
{ type: "pong" }
```

Server → Client Messages

```
{ type: "event", event: Event }
{ type: "session.update", session_id, lifecycle, summary?, stats? }
{ type: "remote.update", remote_env_id, status, public_ip? }
{ type: "ping" }
{ type: "error", message }
{ type: "subscribed", subscription }
{ type: "unsubscribed", subscription }
```

Keepalive

Ping/pong at 30s intervals, 10s pong timeout → stale client terminated after 40s total.

Broadcasting

`createBroadcaster(clients, logger)` — dispatches to clients whose subscriptions match `"all"`, `"workspace:{id}"`, or `"session:{id}"`. Fire-and-forget: failed sends remove the client, never throw.

Consumer integration (consumer.ts:171-184): After successful `processEvent`:

- `broadcastEvent(event)` — always
- `broadcastSessionUpdate(sessionId, workspaceId, "detected")` on `session.start`
- `broadcastSessionUpdate(sessionId, workspaceId, "ended")` on `session.end`

---

16. Authentication & Security

API Authentication

File: `packages/server/src/middleware/auth.ts`

Single API key, Bearer token scheme with **constant-time comparison**:

```
// auth.ts:21-66
const expectedKeyBuffer = Buffer.from(apiKey, "utf-8");
// Per request:
// 1. Check Authorization header + "Bearer " prefix
// 2. Length check (reject before timingSafeEqual if lengths differ)
// 3. crypto.timingSafeEqual(receivedKeyBuffer, expectedKeyBuffer)
```

Applied to all `/api/*` routes **except** `/api/health` (mounted before auth middleware).

WebSocket Authentication

Token query parameter `?token=<api_key>` — plain equality check (`ws/index.ts:99`). Not constant-time (lower risk since WS connections are long-lived, not per-request).

Config Security

`~/.fuel-code/config.yaml` contains the API key. Written with `chmod 0o600` (owner read/write only).

Security Model

- **Single-user auth**: One API key generated at `fuel-code init`
- No user management, no RBAC, no session tokens
- The system is designed for a single developer's personal use

---

17. Local Queue & Offline Resilience

Queue Write

File: `packages/cli/src/lib/queue.ts`

```
~/.fuel-code/
  queue/              — pending events ({ULID}.json)
```

```
  dead-letter/        — events with 100+ failed attempts
  .drain.lock         — PID lockfile for concurrency control
```

Atomic writes: temp file (`.{id}.json.tmp.{hex}`) → rename. ULID filenames ensure chronological sort by filename.

`enqueueEvent()` **NEVER throws** — returns empty string on disk failure. This is the last-resort fallback.

Queue Drain

File: `packages/cli/src/lib/drain.ts`

```
1. listQueuedEvents() → sorted by ULID
2. Batch into groups of config.pipeline.batch_size (default 50)
3. For each batch:
   a. Read events, dead-letter corrupted files
   b. Check _attempts counter, dead-letter if >= 100
   c. POST batch to /api/events/ingest
   d. On success: removeQueuedEvent() for all
   e. On 401: stop immediately (bad API key)
   f. On network/timeout: stop, increment _attempts
4. Return DrainResult { drained, duplicates, remaining, deadLettered, errors }
```

Background Drain

File: `packages/cli/src/lib/drain-background.ts`

- Called by `spawnBackgroundDrain()` after an event is queued
- Spawns `bun -e <inlineScript>` as a detached child process
- 1-second debounce (accumulates rapid hook fires)
- Lockfile (`~/.fuel-code/.drain.lock`) prevents concurrent drains — checks PID liveness via `process.kill(pid, 0)`

---

18. Historical Session Backfill

Files: `packages/cli/src/commands/backfill.ts` , `packages/core/src/session-backfill.ts`

Scan Phase

`scanForSessions()` :

1. Read `~/.claude/projects/` — CC stores one directory per project
2. Per project dir: read `sessions-index.json` for pre-indexed metadata
3. Per `.jsonl` file:
   - Validate UUID format
   - Skip files modified within last 5 minutes (active sessions)
   - Read first 5 lines for `firstTimestamp` , `gitBranch` , `cwd`
   - Read last 64KB for `lastTimestamp`
   - Resolve workspace from `resolvedCwd` via git commands
4. Results sorted by `firstTimestamp` ascending

Path reconstruction (`session-backfill.ts:197-252`): Converts CC project directory names (e.g., `-Users-john-Desktop-my-project`) back to filesystem paths using a greedy algorithm that checks each

accumulated segment against the real filesystem.

Ingest Phase

Concurrent worker pool (default 10):

```
For each session:
  1. Skip if already ingested (resume support)
  2. GET /api/sessions/{id} → skip if 200
  3. POST /api/events/ingest with synthetic session.start event
  4. POST /api/events/ingest with synthetic session.end event
  5. POST /api/sessions/{id}/transcript/upload with retry
     (max 15 attempts, exponential backoff)
     Retries on: 404 (not yet processed), 429 (rate limit), 503, EAGAIN
```

Backfill State

Persisted at `~/.fuel-code/backfill-state.json`:

```
{
  "lastRunAt": "ISO timestamp",
  "lastRunResult": { "ingested": N, "skipped": N, "failed": N },
  "isRunning": false,
  "ingestedSessionIds": ["uuid1", "uuid2"]
}
```

Auto-trigger: Both `fuel-code init` and `fuel-code hooks install` spawn `fuel-code backfill` as a detached background process if sessions are found.

---

19. TUI Architecture (Ink/React)

Entry: `packages/cli/src/tui/App.tsx` → `launchTui()` renders `<App />` via Ink's `render()`.

Component Tree

```
App (App.tsx)
├── Creates FuelApiClient + WsClient from config
├── Connects WS on mount, subscribes { scope: 'all' }
├── Routes: "dashboard" | "session-detail"
│
├── Dashboard (Dashboard.tsx)
│   ├── useWorkspaces(api)       — fetches workspace list
│   ├── useSessions(api, wsId)   — fetches sessions for selected workspace
│   ├── useWsConnection(ws)      — tracks WS state
│   ├── useTodayStats(workspaces)— aggregates lifetime stats
│   ├── WorkspaceItem.tsx        — single workspace in left pane
│   └── StatusBar.tsx            — bottom status strip
│
└── SessionDetailView (SessionDetailView.tsx)
    ├── useSessionDetail(api, ws, id) — fetches session + transcript + git
    ├── SessionHeader.tsx         — metadata bar
```

```
    ├── TranscriptViewer.tsx      — scrollable message list
    ├── Sidebar.tsx               — git/tools/files panels
    │   ├── GitActivityPanel.tsx
    │   ├── ToolsUsedPanel.tsx
    │   └── FilesModifiedPanel.tsx
    ├── MessageBlock.tsx          — single transcript message
    └── FooterBar.tsx             — keybinding help
```

Keybindings (SessionDetailView)

Key	Action
`b` / Escape	Back to dashboard
`t` / `e` / `g`	Switch between transcript/events/git tabs
`j` / `k`	Scroll by message
Space	Page down
`x`	Export session JSON to disk
`q`	Quit

Live Updates

The TUI uses WebSocket for real-time updates rather than polling. `useWsConnection.ts` listens to `WsClient` EventEmitter events (`connected` , `disconnected` , `reconnecting`). The Dashboard applies live `session.update` WS messages via `updateSession()` / `prependSession()` without re-fetching.

---

20. Testing Infrastructure

Docker Compose for Tests

File: `docker-compose.test.yml`

```yaml
services:
  postgres:    # postgres:16-alpine, port 5433, tmpfs for RAM speed
  redis:       # redis:7-alpine, port 6380
  localstack:  # localstack/localstack, S3 only, port 4566
```

Non-standard ports (5433, 6380) to avoid conflicts with local instances.

Test Patterns

Unit tests: `bun:test` with `describe.skipIf(!DATABASE_URL)` to gate database-dependent tests.

E2E pipeline tests (`packages/server/src/__tests__/e2e/pipeline.test.ts`):

```
// Setup:
// 1. createDb() + runMigrations() against port 5433
// 2. Two Redis clients (port 6380): one non-blocking, one for consumer
// 3. flushall() for clean state
```

```
// 4. ensureConsumerGroup()
// 5. startConsumer()
// 6. createApp() + server.listen(0) — random port

// Cleanup per test:
// afterEach: flushall Redis, wait 500ms, TRUNCATE all tables
```

Phase 2 E2E (`phase2-pipeline.test.ts`): Adds LocalStack S3 (port 4566) for full transcript upload → parse → summarize flow.

Mock patterns: S3 client mocked as interface implementation:

```
function createMockS3(transcriptContent: string): S3Client {
  return {
    upload: async (key, body) => ({ key, size: ... }),
    download: async () => transcriptContent,
  };
}
```

`waitFor()` helper: Polls Postgres with configurable timeout/interval until a condition is met (used in E2E tests to wait for async pipeline processing).

---

21. Deployment & Infrastructure

Backend: Railway

- **No Dockerfile exists yet** — Railway auto-detects Bun/Node apps
- Express server, Postgres (Railway managed), Redis (Railway managed)
- Health check at `GET /api/health` (unauthenticated) for Railway health probes
- S3 for blob storage (real AWS, not Railway)

Required Environment Variables

```
DATABASE_URL          — Postgres connection URI
REDIS_URL             — Redis connection URI
API_KEY               — Bearer token for all authenticated endpoints
PORT                  — HTTP port (default 3000)
LOG_LEVEL             — Pino log level (default "info")
NODE_ENV              — "production" suppresses pretty-printing + stack traces
S3_BUCKET             — S3 bucket name (default "fuel-code-blobs")
S3_REGION             — AWS region (default "us-east-1")
S3_ENDPOINT           — Optional LocalStack endpoint
S3_FORCE_PATH_STYLE   — "true" for LocalStack
ANTHROPIC_API_KEY     — For LLM-powered session summaries
```

(`packages/server/src/index.ts:46-71` , `packages/server/.env.example`)

---

22. Architectural Invariants

These are defined in `tasks/CORE.md:1492-1506` and hold across the codebase:

1. **Every event has a workspace_id.** Events outside git repos use `"_unassociated"`.
2. **Every session belongs to exactly one workspace and one device.** Determined at session start, never changes.
3. **Events are immutable.** Once written, never updated or deleted (except archival TTL).
4. **The local queue never drops events.** Events either reach the server or sit in the queue (moved to dead-letter after 100 attempts).
5. **Raw transcripts in S3 are never deleted.** They are the source of truth.
6. **Local and remote devices are architecturally symmetric.** The processing pipeline does not distinguish between them.
7. **Workspace canonical IDs are deterministic.** Given a git remote URL, any machine computes the same canonical ID without a server round-trip.
8. **The server API is the sole interface for data access.** CLI and future web UI both consume the same REST + WebSocket API. No direct DB access from UI packages.
9. **Adding a new view (web, mobile, etc.) requires zero changes to `core/` or `server/`.** Only a new package consuming the API.
10. **The analysis layer (V2) reads from V1 tables and writes to its own tables.** No V1 schema changes needed.

Implementation-Level Invariants (derived from code)

11. **`enqueueEvent()` never throws** — it's the last-resort fallback path.
12. **`runSessionPipeline()` never throws** — all errors are caught and returned as results.
13. **`generateSummary()` never throws** — failure keeps the session at `parsed`, does not regress lifecycle.
14. **Hooks always exit 0** — they must never block Claude Code or git operations.
15. **Consumer never crashes the process** — all errors are caught, logged, and retried.
16. **Session lifecycle transitions use optimistic locking** — `WHERE lifecycle = ANY($from)` prevents concurrent corruption.
17. **Event deduplication is stateless** — `ON CONFLICT (id) DO NOTHING` at both Redis consumer and Postgres insert levels.

---

23. Current Implementation Status vs. CORE.md Spec

Phase	CORE.md Description	Status
Phase 1: Foundation	Events flow from hooks to Postgres	**Complete**
Phase 2: Session Lifecycle	Transcript parsing, summarization, backfill	**Complete**
Phase 3: Git Tracking	Git hooks, git_activity, session-git correlation	**Complete**
Phase 4: CLI + TUI	All commands, TUI dashboard, session detail, WebSocket	**Complete**
Phase 5: Remote Dev Envs	EC2 provisioning, Docker, blueprints, remote up/down	**Not started**
Phase 6: Hardening	Retry logic, progress indicators, archival	**Partially complete**

Phase 5 Gaps (Remote Dev Environments)

The following from CORE.md have no implementation:

- `remote_envs` and `blueprints` Postgres tables (spec exists in CORE.md:523-563, no migration files)
- EC2 provisioning (`aws/ec2.ts` — doesn't exist)
- Docker container setup on EC2
- `fuel-code remote up/down/ssh/ls` commands
- Blueprint auto-detection
- Remote device heartbeat + idle timeout
- SSH key management in S3

Phase 6 Gaps (Hardening)

- Session archival (prune old parsed data) — not implemented
- EC2 tagging for orphan detection — not applicable yet (no remote envs)
- Cost estimation in blueprint output — not applicable yet

Files Specified in CORE.md But Not Yet Created

```
packages/analysis/           — V2 analysis engine (placeholder only)
packages/web/                — V2 web UI (placeholder only)
packages/core/src/blueprint-detector.ts  — auto-detect project environment
infra/docker/Dockerfile.remote
infra/docker/scripts/user-data.sh
infra/railway/railway.toml
infra/sql/schema.sql
packages/server/Dockerfile
```

---

24. Future: V2 Analysis Layer

From `tasks/CORE.md:1311-1358` :

Architecture

```
V1 Tables (read-only for analysis):
  sessions, transcript_messages, content_blocks, events, git_activity
      │
      │ reads from
      ▼
V2 Analysis Engine (packages/analysis/):
  - Prompt extractor: identifies reusable prompts
  - Workflow detector: finds multi-step patterns across sessions
  - Embedding generator: vector embeddings for similarity search
  - Cluster engine: groups similar sessions/prompts/workflows
  - Skill deriver: generates CC skills from recurring patterns
      │
      │ writes to
      ▼
V2 Tables (new, additive):
  analysis_prompts      — extracted prompts with embedding vectors
  analysis_workflows    — multi-step patterns
  analysis_clusters     — groupings in embedding space
```

```
  analysis_skills       — derived Claude Code skills
  analysis_runs         — tracking which analyses have been run
```

Why This Works Without Changing V1

1. `transcript_messages` + `content_blocks` are the rich raw material — already stored for session viewing
2. Sessions have `tags` (TEXT array) that analysis can populate
3. `metadata` JSONB fields on sessions and content_blocks can store annotations
4. New `analysis_*` tables are purely additive

Integration Points

- `server/` adds new routes: `GET /api/analysis/prompts`, `/api/analysis/workflows`, etc.
- CLI/Web adds new views: analysis dashboard, prompt library, skill generator
- No V1 table schema changes required

---

25. Future: Remote Dev Environments (Phase 5)

From `tasks/CORE.md:794-848`:

Provisioning Flow

```
$ fuel-code remote up

1. DETECT / LOAD BLUEPRINT                              (~2s)
   ├── Check .fuel-code/env.yaml in current workspace
   ├── If missing: auto-detect from repo contents
   └── Freeze blueprint → immutable JSON

2. PROVISION EC2 INSTANCE                               (~45-90s)
   ├── Generate ephemeral SSH key pair → upload to S3
   ├── Create/reuse security group (SSH from caller IP only)
   ├── Launch EC2 with Docker-ready AMI + user-data script:
   │   ├── Install Docker, pull image, start container
   │   ├── Clone repo, run setup commands
   │   ├── Install fuel-code CLI + hooks
   │   └── Health check: claude --version
   └── Emit remote.provision.ready event

3. CONNECT                                              (~1s)
   ├── Download ephemeral SSH key from S3
   ├── SSH → exec into Docker container
   └── Events flow back through same pipeline

4. LIFECYCLE
   ├── Auto-terminate after idle timeout (default 60 min)
   ├── Auto-terminate after TTL (default 8 hours)
   └── Manual: fuel-code remote down <id>
```

Device Symmetry

A provisioned remote EC2 is just another Device. It has its own `device_id`, fuel-code installed with hooks, and events flowing through the same pipeline. The backend does **not** special-case remote events.

Planned Database Tables

```
-- remote_envs (CORE.md:523-549)
CREATE TABLE remote_envs (
    id TEXT PRIMARY KEY,
    workspace_id TEXT NOT NULL REFERENCES workspaces(id),
    device_id TEXT REFERENCES devices(id),
    status TEXT NOT NULL DEFAULT 'provisioning'
        CHECK (status IN
('provisioning','ready','active','idle','terminated','error')),
    instance_id TEXT,
    instance_type TEXT NOT NULL,
    region TEXT NOT NULL,
    public_ip TEXT,
    ssh_key_s3_key TEXT,
    blueprint JSONB NOT NULL,
    ttl_minutes INTEGER NOT NULL DEFAULT 480,
    idle_timeout_minutes INTEGER NOT NULL DEFAULT 60,
    ...
);

-- blueprints (CORE.md:554-563)
CREATE TABLE blueprints (
    id TEXT PRIMARY KEY,
    workspace_id TEXT REFERENCES workspaces(id),
    name TEXT NOT NULL,
    source TEXT NOT NULL,      -- "auto-detected" | "manual"
    config JSONB NOT NULL,    -- full env.yaml content
    ...
);
```

---

26. Key Architectural Decisions & Trade-offs

1. Redis Streams as Event Transport (not direct Postgres writes)

Decision: HTTP POST → Redis Stream → Consumer → Postgres **Rationale**: Decouples ingestion latency from processing latency. The HTTP endpoint returns 202 immediately after Redis write. Processing happens asynchronously. This is critical for hook performance — hooks must never block CC or git. **Trade-off**: Adds operational complexity (consumer group management, pending entry reclamation, NOGROUP recovery after Redis restarts). Justified by the non-blocking requirement.

2. Two Redis Clients

Decision: One for HTTP operations, one dedicated to the consumer's blocking `XREADGROUP BLOCK`. **Rationale**: `XREADGROUP BLOCK` holds the connection for up to 5 seconds. If health checks or `XADD` calls share the same connection, they queue behind the block. **Reference**: `packages/server/src/index.ts:115-118`

3. ULID-Based Event Deduplication

Decision: Events carry client-generated ULIDs. `INSERT ... ON CONFLICT (id) DO NOTHING` at both Redis consumer and Postgres levels. **Rationale**: Stateless dedup — no need for a separate "seen events" cache. ULIDs are monotonically increasing, providing chronological ordering by ID. **Trade-off**: Requires clients to generate globally unique IDs. ULIDs from the `ulidx` library provide this with timestamp prefix + random suffix.

4. Optimistic Locking for Lifecycle Transitions

Decision: `UPDATE WHERE lifecycle = ANY($from)` instead of pessimistic locks. **Rationale**: Avoids lock contention between consumer workers. Concurrent transitions to the same state are harmless (second one returns 0 rows). Invalid transitions are detected by querying actual state after failure. **Reference**: `packages/core/src/session-lifecycle.ts:127-219`

5. Summary Failure Doesn't Regress Lifecycle

Decision: If `generateSummary()` fails, the session stays at `parsed` (not `failed`). **Rationale**: The session is fully queryable without a summary. Summaries are best-effort enrichment. A rate limit or API outage shouldn't make the session appear broken. **Reference**: `packages/core/src/session-pipeline.ts:254`

6. `core/` Has Zero Infrastructure Knowledge

Decision: `core/` depends only on `shared`, `postgres`, `pino`, and `@anthropic-ai/sdk`. All S3/Redis/HTTP dependencies are injected via interfaces. **Rationale**: Enables testing with mocks. Enforces that domain logic is reusable (e.g., the analysis layer will import from `core/`). The `S3Client` interface at `session-pipeline.ts:34` is the canonical example. **Trade-off**: Requires wiring code in `server/` (`pipeline/wire.ts`) to connect dependencies.

7. Heuristic Git-Session Correlation

Decision: Link git events to the most recent active session on the same device+workspace. **Rationale**: Works because CC sessions and git operations happen on the same device in the same workspace. No explicit tracking of "which session caused this commit" is needed. **Trade-off**: If two CC sessions are active simultaneously on the same workspace+device, the most recent one wins. The spec acknowledges this is rare and acceptable. **Reference**: `packages/core/src/git-correlator.ts:53-67`

8. Fire-and-Forget Hook Design

Decision: All hooks exit immediately (background subshells, always exit 0). **Rationale**: Hooks must never slow down CC startup (SessionStart) or git operations (post-commit). Even a 100ms delay is unacceptable for git hooks. **Trade-off**: Failures are silent. The local queue + drain mechanism provides eventual delivery, but there's no immediate feedback to the user if an event fails.

9. Single API Key Authentication

Decision: One `fc_...` key for everything. No multi-user, no RBAC. **Rationale**: fuel-code is a personal developer tool. Multi-user adds complexity with no value for the target use case. Can be revisited if the system is ever multi-tenant. **Reference**: `packages/server/src/middleware/auth.ts`

10. Postgres as Primary Query Store, S3 as Durable Blob Store

Decision: Structured data (sessions, events, parsed transcripts) in Postgres. Raw transcripts and large artifacts in S3. Parsed data is recoverable from S3. **Rationale**: Postgres provides rich querying (full-text search on content_blocks via GIN index, array containment on tags). S3 provides cheap durable storage for large blobs. The parsed data in Postgres can be pruned (archival) and re-derived from S3 raw transcripts.

This document was generated by exhaustive inspection of the fuel-code codebase at commit `6cfc158`. All file paths and line references were verified against the source at the time of writing.