The Event Abstraction: Deep Dive

> *Last updated: 2026-02-27. Based on full codebase audit of all event-related source files across all 5 packages.*

Table of Contents

1. What the Abstraction Provides
2. The 14 Event Types
3. The Core Interface and Schema
4. The Payload Registry
5. The 6-Stage Pipeline
6. Stage 1: Hook Layer (Event Generation)
7. Stage 2: CLI Emit (Event Packaging)
8. Stage 3: HTTP Ingest (Server Boundary)
9. Stage 4: Redis Stream (Transport)
10. Stage 5: Consumer (Dispatch Loop)
11. Stage 6: Event Processor + Handlers (Domain Logic)
12. The Local Queue Fallback
13. WebSocket Broadcasting
14. The Database Schema
15. The Handler Registry
16. Individual Handler Deep Dives
17. Cross-Cutting Mechanisms
18. Gap Analysis: What's Missing or Misaligned
19. References

---

1. What the Abstraction Provides

The Event is the **atomic unit of everything** in fuel-code. Every signal that enters the system — a Claude Code session starting, a git commit, a branch checkout — arrives as an Event. Events are immutable, timestamped, typed facts. They are never updated or deleted by application logic.

The Event abstraction spans every package in the monorepo: type definitions in `shared`, processing logic in `core`, HTTP transport in `server`, emission in `cli`, and generation in `hooks`. It is the backbone that all other abstractions build on:

- **Sessions** are created and ended by `session.start` and `session.end` events
- **Git Activity** rows are materialized from `git.commit`, `git.push`, `git.checkout`, `git.merge` events
- **Workspaces** are resolved (created on first contact) during event processing
- **Devices** are resolved (created on first contact) during event processing
- **Transcripts** are triggered by `session.end` events flowing through the pipeline

Core value proposition: A unified, type-safe, validated, durable transport for all activity data. The event pipeline is the single point of entry — there is no other way to get data into the system. This makes the system predictable: to understand what data exists, you only need to understand what events are emitted.

The Event is **one of the original five CORE.md abstractions** (Workspace, Session, Event, Device, Blueprint). CORE.md describes it as "the atomic unit of activity data — immutable, append-only, timestamped."

2. The 14 Event Types

Defined in `packages/shared/src/types/event.ts` :

```
export type EventType =
  | "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";
```

The types follow a dot-notation hierarchy: `domain.action` .

Implementation Status by Type

Type	Zod Schema	Hook Emitter	Handler	Materialized Output
`session.start`	Yes	`SessionStart.sh` → `session-start.ts`	`handleSessionStart`	`sessions` row (lifecycle=detected)
`session.end`	Yes	`SessionEnd.sh` → `session-end.ts`	`handleSessionEnd`	session transition → transcript pipeline
`session.compact`	Yes	**No hook exists**	**No handler**	Persisted to `events` only
`git.commit`	Yes	`post-commit` (bash)	`handleGitCommit`	`git_activity` row (type=commit)
`git.push`	Yes	`pre-push` (bash)	`handleGitPush`	`git_activity` row (type=push)
`git.checkout`	Yes	`post-checkout` (bash)	`handleGitCheckout`	`git_activity` row (type=checkout)
`git.merge`	Yes	`post-merge` (bash)	`handleGitMerge`	`git_activity` row (type=merge)
`remote.provision.start`	**No**	**No**	**No**	Forward declaration (Phase 5)

`remote.provision.ready`	**No**	**No**	**No**	Forward declaration (Phase 5)
`remote.provision.error`	**No**	**No**	**No**	Forward declaration (Phase 5)
`remote.terminate`	**No**	**No**	**No**	Forward declaration (Phase 5)
`system.device.register`	**No**	**No**	**No**	Forward declaration
`system.hooks.installed`	**No**	**No**	**No**	Forward declaration
`system.heartbeat`	**No**	**No**	**No**	Forward declaration

Summary: 7 of 14 types have schemas. 6 of 14 have handlers. 6 of 14 have hook emitters. 7 types are pure forward declarations with no implementation beyond the type definition.

Category Breakdown

`session.*` (3 types) — Claude Code session lifecycle:

- `session.start` : CC session began. Creates a session record.
- `session.end` : CC session ended. Transitions session state, triggers transcript processing.
- `session.compact` : CC context window was compacted. Has a Zod schema but no hook emitter and no handler.

`git.*` (4 types) — Git operations detected by bash hooks:

- `git.commit` : Post-commit hook. Richest payload (hash, message, author, branch, diff stats, file list).
- `git.push` : Pre-push hook. The only hook that fires *before* the git operation.
- `git.checkout` : Post-checkout hook. Only branch checkouts (not file checkouts).
- `git.merge` : Post-merge hook. Detects merged branch and conflicts.

`remote.*` (4 types) — Remote dev environment provisioning (Phase 5, unimplemented):

- `remote.provision.start` , `remote.provision.ready` , `remote.provision.error` , `remote.terminate` .

`system.*` (3 types) — Infrastructure signals (unimplemented):

- `system.device.register` , `system.hooks.installed` , `system.heartbeat` .

---

3. The Core Interface and Schema

3.1 The `Event` Interface

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

```
export interface Event {
  id: string;                    // ULID — globally unique, time-sortable
  type: EventType;               // One of 14 types
  timestamp: string;             // ISO-8601, client-side (originating machine's
clock)
```

```
  device_id: string;           // Originating device ULID
  workspace_id: string;        // Workspace canonical ID (e.g.,
"github.com/user/repo")
  session_id: string | null;   // CC session ID (null for git hooks outside
sessions)
  data: Record<string, unknown>; // Type-specific payload
  ingested_at: string | null;  // Server-side timestamp (null before ingestion)
  blob_refs: BlobRef[];        // S3 references for large payloads
}
```

Key design decisions:

Decision	Rationale
`id` is a ULID	Time-sortable, client-generated (no server round-trip), collision-free
`workspace_id` is canonical string on wire, ULID in DB	The CLI sends `"github.com/user/repo"`, the processor resolves to ULID before INSERT
`session_id` is nullable	Git operations can happen outside CC sessions (terminal commits, CI pushes)
`data` is `Record<string, unknown>`	Type-specific payloads vary by event type; validated at ingest time by Zod schemas
`blob_refs` for large payloads	Events stay small; transcripts and large diffs live in S3
`ingested_at` is server-set	Client clocks can't be trusted for ordering; `ingested_at` provides a server-authoritative timestamp

3.2 The Envelope Schema (Zod)

`packages/shared/src/schemas/event-base.ts` :

```
const ULID_REGEX = /^[0-9A-HJKMNP-TV-Z]{26}$/;

const blobRefSchema = z.object({
  key: z.string().min(1),
  content_type: z.string().min(1),
  size_bytes: z.number().int().nonnegative(),
});

const eventSchema = z.object({
  id: z.string().regex(ULID_REGEX, "Must be a valid ULID"),
  type: z.enum(EVENT_TYPES),
  timestamp: z.string().datetime(),
  device_id: z.string().min(1),
  workspace_id: z.string().min(1),
  session_id: z.string().nullable().default(null),
  data: z.record(z.unknown()).default({}),
  blob_refs: z.array(blobRefSchema).default([]),
});
```

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

Validation layers:

1. **Envelope validation** at the HTTP boundary — `ingestRequestSchema.parse()` validates structure
2. **Payload validation** after Redis consumption — `validateEventPayload()` validates `data` per-type
3. **Handler-level extraction** — handlers cast fields from `event.data` without using validated types

3.3 The `IngestRequest` / `IngestResponse` Envelope

```
interface IngestRequest {
  events: Event[];  // Batch of 1–100 events
}

interface IngestResponse {
  ingested: number;    // Successfully accepted
  duplicates: number;  // Skipped (already seen by ULID)
  rejected?: number;   // Failed validation
  errors?: Array<{ index: number; error: string }>;  // Per-event errors
}
```

---

4. The Payload Registry

`packages/shared/src/schemas/payload-registry.ts`:

The payload registry maps event types to Zod schemas for type-specific `data` validation:

```
const PAYLOAD_SCHEMAS: Partial<Record<EventType, z.ZodSchema>> = {
  "session.start": sessionStartPayloadSchema,
  "session.end": sessionEndPayloadSchema,
  "session.compact": sessionCompactPayloadSchema,
  "git.commit": gitCommitPayloadSchema,
  "git.push": gitPushPayloadSchema,
  "git.checkout": gitCheckoutPayloadSchema,
  "git.merge": gitMergePayloadSchema,
};

function validateEventPayload(type: EventType, data: unknown) {
  const schema = PAYLOAD_SCHEMAS[type];
  if (!schema) return { success: true, data };  // No schema = passthrough
  return schema.safeParse(data);
}
```

7 schemas registered, 7 types with no schema — unregistered types pass validation unconditionally.

4.1 Session Payload Schemas

`session.start` (`packages/shared/src/schemas/session-start.ts`):

Field	Type	Description
`cc_session_id`	`string` (min 1)	Claude Code's own session UUID
`cwd`	`string` (min 1)	Working directory at session start
`git_branch`	`string | null`	Current git branch (null if not a git repo)
`git_remote`	`string | null`	Git remote URL for workspace resolution
`cc_version`	`string | null`	Claude Code version (e.g., "2.1.47")
`model`	`string | null`	Model being used (e.g., "claude-opus-4-6")
`source`	`enum`	`"startup" | "resume" | "clear" | "compact" | "backfill"`
`transcript_path`	`string`	Path to CC's JSONL transcript file on disk

`session.end` (`packages/shared/src/schemas/session-end.ts`):

Field	Type	Description
`cc_session_id`	`string` (min 1)	Claude Code's session UUID
`duration_ms`	`number` (int, >= 0)	Session duration in milliseconds
`end_reason`	`enum`	`"exit" | "clear" | "logout" | "crash"`
`transcript_path`	`string`	Path to JSONL for upload

`session.compact` (`packages/shared/src/schemas/session-compact.ts`):

Field	Type	Description
`cc_session_id`	`string` (min 1)	Claude Code's session UUID
`compact_sequence`	`number` (int, >= 0)	0 = original, 1 = first compact, etc.
`transcript_path`	`string`	Path to JSONL for mid-session backup

4.2 Git Payload Schemas

`git.commit` (`packages/shared/src/schemas/git-commit.ts`):

Field	Type	Description
`hash`	`string`	Full SHA from `git rev-parse HEAD`
`message`	`string`	Commit message (first 8192 chars)
`author_name`	`string`	Commit author name

`author_email`	`string` (optional)	Commit author email
`branch`	`string`	Current branch
`files_changed`	`number` (int, >= 0)	From `git diff-tree --numstat`
`insertions`	`number` (int, >= 0)	Lines added
`deletions`	`number` (int, >= 0)	Lines removed
`file_list`	`array` (optional)	`[{ path: string, status: string }]` — A/M/D/R per file

`git.push` (`packages/shared/src/schemas/git-push.ts`):

Field	Type	Description
`branch`	`string`	Branch being pushed
`remote`	`string`	Remote name (e.g., "origin")
`commit_count`	`number` (int, >= 0)	Number of commits being pushed
`commits`	`string[]` (optional)	Array of pushed commit SHAs (max 100)

`git.checkout` (`packages/shared/src/schemas/git-checkout.ts`):

Field	Type	Description
`from_ref`	`string`	Source commit SHA
`to_ref`	`string`	Target commit SHA
`from_branch`	`string | null`	Source branch name (null for detached HEAD)
`to_branch`	`string | null`	Target branch name (null for detached HEAD)

`git.merge` (`packages/shared/src/schemas/git-merge.ts`):

Field	Type	Description
`merge_commit`	`string`	The merge commit SHA
`message`	`string`	Merge commit message (first 4096 chars)
`merged_branch`	`string`	The branch that was merged in
`into_branch`	`string`	The branch that received the merge
`files_changed`	`number` (int, >= 0)	Files changed in the merge
`had_conflicts`	`boolean`	Whether conflicts were detected

---

5. The 6-Stage Pipeline

Every event passes through 6 stages from creation to persistence:

```
Stage 1: Hook Layer        — Detects the activity, extracts metadata
Stage 2: CLI Emit          — Packages into Event envelope, POSTs to backend
Stage 3: HTTP Ingest       — Validates envelope, publishes to Redis Stream
Stage 4: Redis Stream      — Durable transport with consumer groups
Stage 5: Consumer          — Reads from stream, dispatches to processor
Stage 6: Event Processor   — Resolves entities, persists, dispatches to handler
```

With a parallel fallback path:

```
Stage 2 (failure) → Local Queue → Background Drain → Stage 3 (retry)
```

And a broadcast path:

```
Stage 5 (after success) → WebSocket Broadcaster → Connected CLI Clients
```

---

6. Stage 1: Hook Layer (Event Generation)

Events originate from two hook systems:

6.1 Claude Code Hooks (TypeScript)

Location: `packages/hooks/claude/`

Two hooks emit session lifecycle events:

`SessionStart.sh` → `_helpers/session-start.ts`

1. Reads CC hook context from stdin (JSON with `session_id`, `cwd`, `transcript_path`)
2. Resolves workspace: calls `resolveWorkspace(cwd)` which runs `git remote get-url origin`, normalizes the URL
3. Gets CC version from `claude --version` (3s timeout, best-effort)
4. Constructs `session.start` payload:

   ```
   {
     cc_session_id, cwd, git_branch, git_remote,
     cc_version, model: null, source: "startup",
     transcript_path
   }
   ```

5. Calls `fuel-code emit session.start --workspace-id <canonical> --session-id <cc_session_id> --data-stdin`
6. Fire-and-forget in background (the `.sh` wrapper pipes stdin to the TS helper and exits immediately)

`SessionEnd.sh` → `_helpers/session-end.ts`

1. Reads CC hook context from stdin
2. Resolves workspace
3. Constructs `session.end` payload with `duration_ms: 0` (server computes actual duration from `started_at`)
4. Calls `fuel-code emit session.end --workspace-id <canonical> --session-id <cc_session_id> --data-stdin`

5. After emit, spawns background `fuel-code transcript upload --session-id <id> --file <path>`

`_helpers/resolve-workspace.ts` — Shared workspace resolution:

- Checks if CWD is a git repo (`git rev-parse --is-inside-work-tree`)
- Gets current branch (`git branch --show-current`)
- Tries `origin` remote, falls back to first remote alphabetically
- If no remote: uses `local:<sha256(first-commit-hash)>`
- If not a git repo at all: returns `_unassociated`
- Never throws — defaults to `_unassociated` on any failure

6.2 Git Hooks (Bash)

Location: `packages/hooks/git/`

Four hooks emit git activity events:

`post-commit` — Fires after `git commit` :

```bash
# Extract metadata
COMMIT_HASH=$(git rev-parse HEAD)
COMMIT_MSG=$(git log -1 --format='%B' HEAD | head -c 8192)
AUTHOR_NAME=$(git log -1 --format='%an' HEAD)
AUTHOR_EMAIL=$(git log -1 --format='%ae' HEAD)
BRANCH=$(git branch --show-current 2>/dev/null || git rev-parse --short HEAD)

# Diff stats via git diff-tree --numstat
FILES_CHANGED=... INSERTIONS=... DELETIONS=...

# File list: [{path, status}]
FILE_LIST=$(git diff-tree --no-commit-id --name-status -r HEAD)

# Emit in background
echo "$JSON_PAYLOAD" | fuel-code emit git.commit \
  --workspace-id "$WORKSPACE_ID" --data-stdin &
```

`pre-push` — Fires before `git push` :

```bash
# MUST consume all stdin before doing anything else
while IFS=' ' read -r local_ref local_sha remote_ref remote_sha; do
  PUSH_REFS+="..."
done

# Dispatch to local hook first (can block the push)
LOCAL_HOOK="$REPO_GIT_DIR/hooks/pre-push"
if [ -x "$LOCAL_HOOK" ]; then
  echo -e "$PUSH_REFS" | "$LOCAL_HOOK" "$@"
  LOCAL_EXIT=$?
  [ $LOCAL_EXIT -ne 0 ] && exit $LOCAL_EXIT  # Respect local hook's decision
fi

# Extract: branch, remote, commit_count, commits array
```

```
echo "$JSON_PAYLOAD" | fuel-code emit git.push \
  --workspace-id "$WORKSPACE_ID" --data-stdin &
```

The pre-push hook is the **only hook that can block a git operation** — but only by delegating to the repo's local `pre-push` hook. fuel-code's own logic always exits 0.

`post-checkout` — Fires after branch switch:

```
# Only track branch checkouts ($3 == 1), not file checkouts
[ "$3" != "1" ] && exit 0

# Extract: from_ref ($1), to_ref ($2), from_branch, to_branch
echo "$JSON_PAYLOAD" | fuel-code emit git.checkout \
  --workspace-id "$WORKSPACE_ID" --data-stdin &
```

`post-merge` — Fires after merge:

```
# Detect merged branch from MERGE_HEAD or commit message
MERGE_HEAD=$(cat "$(git rev-parse --git-dir)/MERGE_HEAD" 2>/dev/null)
MERGED_BRANCH=$(git log -1 --format='%s' HEAD | grep -oP "Merge branch '\\K[^']+")

# Detect conflicts
HAD_CONFLICTS=$(grep -q "Conflicts:" "$(git rev-parse --git-dir)/MERGE_MSG" && echo
true || echo false)

echo "$JSON_PAYLOAD" | fuel-code emit git.merge \
  --workspace-id "$WORKSPACE_ID" --data-stdin &
```

6.3 Shared Hook Safety Invariants

All hooks follow these safety rules:

```
1. ALWAYS exit 0 (except pre-push delegating to a local hook)
2. Fire-and-forget: fuel-code emit is backgrounded with &
3. Never block the user's git/CC operation
4. Errors are logged to ~/.fuel-code/hook-errors.log
5. Per-repo opt-out via .fuel-code/config.yaml → git_enabled: false
6. Check for fuel-code in PATH before attempting emit
```

6.4 Hook Chaining

Every git hook dispatches in order:

```
# 1. Repo-local hook (.git/hooks/) — forwarded since core.hooksPath overrides it
LOCAL_HOOK="$REPO_GIT_DIR/hooks/post-commit"
if [ -x "$LOCAL_HOOK" ] && ! head -5 "$LOCAL_HOOK" | grep -q "fuel-code:"; then
  "$LOCAL_HOOK" "$@" || true  # Swallow errors (except pre-push)
fi

# 2. User's previous global hook (renamed to *.user during install)
USER_HOOK="$(dirname "$0")/post-commit.user"
```

```
if [ -x "$USER_HOOK" ]; then
  "$USER_HOOK" "$@" || true
fi

# 3. fuel-code's own logic
```

The `grep -q "fuel-code:"` check prevents recursion if someone copies a fuel-code hook into `.git/hooks/`.

---

7. Stage 2: CLI Emit (Event Packaging)

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

The `fuel-code emit <event-type>` command, called by hooks. This is the most performance-critical code path — it must complete in <2 seconds to avoid delaying the user's git operations.

7.1 Flow

```
1. Load config from ~/.fuel-code/config.yaml (best-effort)
2. Parse event data:
   --data '{"key":"value"}'     → JSON from argument
   --data-stdin                 → JSON from stdin
   Invalid JSON                 → wrapped as { _raw: theString }
3. Inject device hints into event.data:
   data._device_name = config.device.name
   data._device_type = config.device.type
4. Construct Event:
   {
     id: generateId(),           // Fresh ULID
     type: <event-type>,
     timestamp: new Date().toISOString(),
     device_id: config.device.id,
     workspace_id: --workspace-id flag,
     session_id: --session-id flag || null,
     data: <parsed data with hints>,
     ingested_at: null,
     blob_refs: []
   }
5. POST to backend:
   POST {config.backend.url}/api/events/ingest
   Body: { events: [event] }
   Timeout: config.pipeline.post_timeout_ms (default 5000ms)
6. On failure: enqueueEvent(event)  → disk queue
7. On success with queued events: spawnBackgroundDrain()
8. EXIT 0 (always)
```

7.2 Key Design Decisions

Always exit 0: The emit command never fails with a non-zero exit code. If config is missing, the backend is down, or the event is malformed — it either queues the event or silently drops it. This ensures hooks never interfere with the user's workflow.

Device hint injection: The CLI injects `_device_name` and `_device_type` as underscore-prefixed fields into `event.data`. These are extracted by the event processor, used for device resolution, then stripped before persistence. This is an untyped mutation of the event payload — a pragmatic hack documented in CORE.md as a known pattern.

Background drain trigger: If the POST succeeds and there are queued events from prior failures, a background drain process is spawned to flush the queue. This piggybacks connectivity recovery on successful emissions.

Invalid JSON handling: If stdin contains unparseable JSON (e.g., a bash variable expansion failure), it's wrapped as `{ _raw: "the raw string" }` rather than failing. This preserves the signal even when the payload is malformed.

---

8. Stage 3: HTTP Ingest (Server Boundary)

`packages/server/src/routes/events.ts`

8.1 Endpoint: `POST /api/events/ingest`

The HTTP boundary between clients (CLI, future web) and the processing pipeline:

```
1. Authenticate (Bearer token from Authorization header)
2. Parse body with ingestRequestSchema.parse() → 400 on failure
3. For each event in the batch:
   a. Validate type-specific payload via validateEventPayload()
   b. If validation fails: mark as rejected, continue to next
   c. If validation passes: set ingested_at = now(), mark as accepted
4. Publish accepted events to Redis Stream via publishBatchToStream()
5. On Redis failure: return 503 with retry_after_seconds: 30
6. Return 202 with per-event results
```

8.2 Response Format

```json
{
  "results": [
    { "id": "01HXYZ...", "status": "accepted" },
    { "id": "01HABC...", "status": "rejected", "error": "Invalid payload" }
  ],
  "ingested": 1,
  "rejected": 1,
  "errors": [{ "index": 1, "error": "data.hash: Required" }]
}
```

8.3 Batch Semantics

- **Batch size**: 1-100 events per request (enforced by Zod schema)
- **Partial success**: Individual events can fail without rejecting the batch
- **Idempotency**: Events with duplicate ULIDs are accepted at HTTP but deduplicated at INSERT (`ON CONFLICT DO NOTHING`)
- **Ordering**: Events within a batch are published to Redis in order, but Redis consumer processing is not guaranteed to preserve that order

9. Stage 4: Redis Stream (Transport)

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

Redis Streams provide durable, at-least-once delivery between the HTTP layer and the consumer.

9.1 Stream Configuration

Constant	Value	Purpose
`EVENTS_STREAM`	`"events:incoming"`	The Redis Stream key
`CONSUMER_GROUP`	`"event-processors"`	Consumer group name
`CONSUMER_NAME`	`"${hostname}-${pid}"`	Unique per-process consumer

9.2 Serialization

Events are flattened into Redis hash-like key-value pairs for `XADD`:

```
function serializeEvent(event: Event): Record<string, string> {
  return {
    id: event.id,
    type: event.type,
    timestamp: event.timestamp,
    device_id: event.device_id,
    workspace_id: event.workspace_id,
    session_id: event.session_id ?? "",      // Empty string for null
    data: JSON.stringify(event.data),         // JSON-encoded
    blob_refs: JSON.stringify(event.blob_refs), // JSON-encoded
    ingested_at: event.ingested_at ?? "",
  };
}
```

Deserialization reverses: parses JSON fields, converts empty strings back to null.

9.3 Key Operations

Publishing (`publishBatchToStream`):

- Uses Redis pipeline for batch efficiency (one round-trip for N events)
- Each `XADD` uses auto-generated Redis stream IDs
- Returns per-event success/failure in `BatchPublishResult`

Reading (`readFromStream`):

- `XREADGROUP GROUP event-processors consumer-name COUNT 10 BLOCK 5000 STREAMS events:incoming >`
- The `>` means "only new messages not yet delivered to this group"
- Blocks up to 5 seconds if no new events (prevents busy-looping)

Acknowledging (`acknowledgeEntry`):

- `XACK events:incoming event-processors <stream-id>`

- Removes the entry from the PEL (Pending Entries List)

Reclaiming (`claimPendingEntries`):

- Tries `XAUTOCLAIM` (Redis 6.2+) first: auto-claims entries idle for >60 seconds
- Falls back to `XPENDING` + `XCLAIM` for older Redis versions
- Used on startup to recover entries from crashed consumers

9.4 Consumer Group Lifecycle

```
Server starts
  → ensureConsumerGroup()
    → XGROUP CREATE events:incoming event-processors $ MKSTREAM
    → If BUSYGROUP: group already exists, ignore
  → claimPendingEntries()
    → Recover stale entries from crashed consumers
  → Main loop begins
```

If Redis restarts and the group is lost (`NOGROUP` error), the consumer recreates it and continues.

---

10. Stage 5: Consumer (Dispatch Loop)

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

The consumer is the main loop that bridges Redis Streams and the event processor.

10.1 Configuration

Constant	Value	Purpose
`READ_COUNT`	10	Events per `XREADGROUP` call
`BLOCK_MS`	5000	Max time to block waiting for events
`CLAIM_IDLE_MS`	60000	Reclaim entries idle >60s
`CLAIM_COUNT`	100	Max entries to reclaim at once
`MAX_RETRIES`	3	Attempts before dead-lettering
`RECONNECT_DELAY`	5000	Wait time on Redis disconnect
`STATS_INTERVAL`	60000	Log stats every 60s
`STOP_TIMEOUT`	10000	Grace period for shutdown

10.2 Main Loop

```
while (!stopped) {
  1. readFromStream(redis, 10, 5000)
     → Returns 0-10 entries (blocks up to 5s if empty)

  2. For each entry:
     a. Deserialize event from Redis fields
```

```
      b. processEvent(sql, event, registry, logger, pipelineDeps)
      c. On success/duplicate:
         - acknowledgeEntry(redis, streamId)
         - broadcastEvent(event) via WebSocket
         - broadcastSessionUpdate() for session.start/session.end
      d. On failure:
         - Increment retryMap[eventId]
         - If retries >= 3: dead-letter (ack + log error)
         - If retries < 3: leave un-acked for re-delivery

   3. On NOGROUP error (Redis restarted):
      - Recreate consumer group
      - Continue immediately

   4. On Redis connection error:
      - Wait RECONNECT_DELAY (5s)
      - Retry ensureConsumerGroup()
}
```

10.3 Startup Recovery

Before entering the main loop, the consumer:

1. **Retries `ensureConsumerGroup()`** until Redis is available (with 5s delay between attempts)
2. **Claims pending entries** via `claimPendingEntries()` — reclaims events that were delivered to a previous consumer process that crashed before acknowledging
3. **Processes reclaimed entries** before reading new ones — ensures crash recovery happens immediately

10.4 Stats Logging

Every 60 seconds, the consumer logs:

```json
{
  "processed": 142,
  "duplicates": 3,
  "errors": 1,
  "deadLettered": 0,
  "uptime": "3600s"
}
```

10.5 Graceful Shutdown

On `stop()`:

1. Set `stopped = true` (exits main loop at next iteration)
2. Wait up to `STOP_TIMEOUT` (10s) for current batch to complete
3. Log final stats

---

11. Stage 6: Event Processor + Handlers (Domain Logic)

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

The event processor is the core function that the consumer calls for each event. It's pure domain logic — no HTTP, no CLI, no UI knowledge.

11.1 Processing Steps

```
processEvent(sql, event, registry, logger, pipelineDeps?)
  │
  ├─ 1. Resolve workspace
  │     event.workspace_id = "github.com/user/repo"  (canonical string)
  │        → resolveOrCreateWorkspace(sql, canonical, hints?)
  │        → Returns ULID: "01HXYZ..."
  │
  ├─ 2. Resolve device
  │     event.device_id = "01HDEV..."
  │        → resolveOrCreateDevice(sql, deviceId, hints?)
  │        → Upserts device row, returns ULID
  │
  ├─ 3. Link workspace ↔ device
  │     → ensureWorkspaceDeviceLink(sql, workspaceUlid, deviceId, cwd)
  │
  ├─ 4. Strip transport hints
  │     → delete event.data._device_name
  │     → delete event.data._device_type
  │
  ├─ 5. Insert event row
  │     INSERT INTO events (...) VALUES (...)
  │     ON CONFLICT (id) DO NOTHING
  │     │
  │     ├─ No rows returned → duplicate, return early
  │     └─ Row returned → continue
  │
  ├─ 6. Validate payload
  │     validateEventPayload(event.type, event.data)
  │     │
  │     ├─ No schema for type → passthrough (success)
  │     ├─ Validation fails → log warning, skip handler
  │     └─ Validation passes → continue
  │
  └─ 7. Dispatch to handler
        registry.getHandler(event.type)
        │
        ├─ No handler → log debug, done
        └─ Handler found → call with EventHandlerContext
              │
              ├─ Success → return processed
              └─ Error → log error, return processed (event already persisted)
```

11.2 The `EventHandlerContext`

```
interface EventHandlerContext {
  sql: Sql;                    // postgres.js client
  event: Event;                 // The event being processed
  workspaceId: string;          // Resolved ULID (not canonical)
  logger: Logger;               // Pino logger scoped to this event
  pipelineDeps?: PipelineDeps; // For post-processing (transcript pipeline)
}
```

11.3 Error Isolation

Handler errors are **logged but never propagate** — the event row is already persisted before the handler runs. This means:

- A bug in `handleGitCommit` doesn't prevent the event from being stored
- The `git_activity` row might not get created, but the raw event data is always in Postgres
- The consumer sees the overall process as "processed" (not "error") even if the handler failed

The only way an event gets status "error" at the processor level is if workspace/device resolution fails — these are infrastructure errors, not handler errors.

11.4 The `ProcessResult`

```
interface ProcessResult {
  eventId: string;
  status: "processed" | "duplicate" | "error";
  handlerResults: Array<{
    type: string;
    success: boolean;
    error?: string;
  }>;
}
```

---

12. The Local Queue Fallback

When the backend is unreachable, events are written to disk and retried later.

12.1 Queue Architecture

Directory: `~/.fuel-code/queue/` (events awaiting delivery) **Dead letter**: `~/.fuel-code/dead-letter/` (permanently failed events)

12.2 Enqueue (`packages/cli/src/lib/queue.ts`)

```
async function enqueueEvent(event: Event, queueDir?: string): Promise<void> {
  // 1. Ensure queue directory exists
  // 2. Write to temp file (.tmp suffix)
  // 3. Atomic rename to final path (ULID-based filename)
  // NEVER throws — silently drops if disk write fails
}
```

Filename format: `{ULID}.json` — ULIDs sort chronologically, so `ls` lists events in order.

Atomicity: Write to `{ULID}.json.tmp` then `rename()` to `{ULID}.json`. Prevents partial reads during concurrent access.

12.3 Drain (`packages/cli/src/lib/drain.ts`)

```
async function drainQueue(config: FuelConfig): Promise<DrainResult> {
  const files = await listQueuedEvents();      // Sorted by ULID (chronological)

  for (const batch of chunk(files, config.pipeline.batch_size)) {
    const events = batch.map(f => readQueuedEvent(f));

    // Track per-event retry count
    for (const event of events) {
      event.data._attempts = (event.data._attempts ?? 0) + 1;
      if (event.data._attempts > 100) {
        await moveToDeadLetter(filePath);
        continue;
      }
    }

    // POST batch to backend
    const result = await api.ingest({ events });

    // On success: delete queued files
    // On 401: stop (bad API key)
    // On 503/timeout: stop, leave remaining
    // On per-event rejection: dead-letter that specific event
  }

  return { drained, duplicates, remaining, deadLettered, errors };
}
```

Retry semantics:

- Each event gets up to 100 attempts (tracked via `_attempts` field in event data)
- After 100 attempts: moved to dead-letter directory
- Corrupted files (unparseable JSON): dead-lettered immediately
- 401 errors: stop immediately (API key is wrong, retrying won't help)
- 503/timeout: stop gracefully, leave remaining events for next drain

12.4 Background Drain (`packages/cli/src/lib/drain-background.ts`)

```
function spawnBackgroundDrain(config: FuelConfig): void {
  // Spawn detached bun process:
  //   1. Sleep 1s (debounce multiple rapid triggers)
  //   2. Acquire lockfile (~/.fuel-code/drain.lock)
  //   3. Run drainQueue(config)
  //   4. Release lockfile
  //
  // Lockfile prevents concurrent drain processes.
  // Stale locks (dead PID) are cleaned up automatically.
}
```

The background drain is triggered:

1. After a successful `fuel-code emit` when queued events exist
2. Explicitly via `fuel-code drain` command

---

13. WebSocket Broadcasting

After successful event processing, the consumer broadcasts to connected WebSocket clients.

13.1 Broadcaster (`packages/server/src/ws/broadcaster.ts`)

```
interface WsBroadcaster {
  broadcastEvent(event: Event): void;
  broadcastSessionUpdate(sessionId, workspaceId, lifecycle, summary?, stats?): void;
  broadcastRemoteUpdate(remoteEnvId, workspaceId, status, publicIp?): void;
}
```

Subscription matching: A client receives a broadcast if:

- It has subscribed to `"all"` , OR
- It has subscribed to `workspace:<workspace_id>` and the event's workspace matches, OR
- It has subscribed to `session:<session_id>` and the event's session matches

Non-blocking: `ws.send()` is fire-and-forget. If a send fails, the client is removed from the set.

13.2 What Gets Broadcast

After the consumer processes an event:

Event Type	Broadcasts
All types	`{ type: "event", event }`
`session.start`	`{ type: "session.update", session_id, workspace_id, lifecycle: "detected" }`
`session.end`	`{ type: "session.update", session_id, workspace_id, lifecycle: "ended" }`

The session update broadcasts carry lifecycle state so TUI clients can update session status without re-fetching.

13.3 WebSocket Protocol (`packages/shared/src/types/ws.ts`)

Client → Server:

Message Type	Purpose	Fields
`subscribe`	Start receiving events	`scope: "all" | workspace_id | session_id`
`unsubscribe`	Stop receiving events	`workspace_id? | session_id?`

`pong`	Keepalive response	(none)

Server → Client:

Message Type	Purpose	Fields
`event`	New event processed	`event: Event`
`session.update`	Session lifecycle changed	`session_id, lifecycle, summary?, stats?`
`remote.update`	Remote env status changed	`remote_env_id, status, public_ip?`
`ping`	Keepalive probe	(none)
`error`	Error notification	`message: string`
`subscribed`	Subscribe acknowledgement	`scope`
`unsubscribed`	Unsubscribe acknowledgement	(none)

13.4 WebSocket Server (`packages/server/src/ws/index.ts`)

- **Path**: `/api/ws`
- **Auth**: `?token=<api_key>` query parameter. Rejects with close code 4001 on failure.
- **Keepalive**: 30s ping interval, 10s pong timeout. Clients that don't respond are disconnected.
- **Client tracking**: Each client gets a ULID ID, a `Set<string>` of subscriptions, and an `isAlive` flag for keepalive.

13.5 CLI WebSocket Client (`packages/cli/src/lib/ws-client.ts`)

`WsClient` extends `EventEmitter` :

- **Connect**: Opens WebSocket to `wss://<backend>/api/ws?token=<key>` . Defers resolution by 50ms to catch immediate auth rejection.
- **Auto-reconnect**: Exponential backoff with jitter: `min(1000 * 2^attempt, maxDelay) + random(0, 500)` . Default max 10 attempts.
- **Emitted events**: `event` , `session.update` , `remote.update` , `connected` , `disconnected` , `reconnecting` , `error` .
- **Subscription persistence**: Subscriptions are stored locally and re-sent on reconnect.

---

14. The Database Schema

14.1 `events` Table

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

```sql
CREATE TABLE events (
    id              TEXT PRIMARY KEY,
    type            TEXT NOT NULL,
    timestamp       TIMESTAMPTZ NOT NULL,
    device_id       TEXT NOT NULL REFERENCES devices(id),
    workspace_id    TEXT NOT NULL REFERENCES workspaces(id),
    session_id      TEXT REFERENCES sessions(id),
    data            JSONB NOT NULL,
    blob_refs       JSONB NOT NULL DEFAULT '[]',
    ingested_at     TIMESTAMPTZ NOT NULL DEFAULT now()
);
```

14.2 Indexes

```
CREATE INDEX idx_events_workspace_time ON events(workspace_id, timestamp DESC);
CREATE INDEX idx_events_session        ON events(session_id, timestamp ASC) WHERE
session_id IS NOT NULL;
CREATE INDEX idx_events_type           ON events(type, timestamp DESC);
CREATE INDEX idx_events_device         ON events(device_id, timestamp DESC);
```

Index	Purpose
`idx_events_workspace_time`	Timeline queries: "all events in this workspace, newest first"
`idx_events_session`	Session event list: "all events in this session, oldest first" (partial — excludes orphans)
`idx_events_type`	Type filtering: "all git.commit events, newest first"
`idx_events_device`	Device event list: "all events from this device, newest first"

Design decisions:

- **No GIN index on `data` JSONB**: Type-specific queries that need JSONB fields go through the `git_activity` table (materialized view) instead
- **Partial index on `session_id`** : Only indexes non-NULL values, saving space for orphan events (git hooks outside sessions)
- **`timestamp DESC` vs `ASC`** : Session events are `ASC` (chronological within a session) while all others are `DESC` (newest first for browsing)

14.3 Column Utilization

Column	Set By	Used By
`id`	CLI (ULID)	Primary key, dedup via `ON CONFLICT`
`type`	CLI (from hook)	Handler dispatch, index filtering
`timestamp`	CLI (client clock)	Ordering, timeline, session correlation
`device_id`	CLI (from config)	Device resolution, git correlation
`workspace_id`	Event processor (resolved from canonical)	Workspace scoping, timeline
`session_id`	CLI (from hook context, nullable)	Session scoping, partial index
`data`	CLI (from hook, hints stripped by processor)	Handler payload, API responses

`blob_refs`	CLI (always `[]` currently)	**Never populated** — S3 refs not implemented
`ingested_at`	Event processor (`new Date().toISOString()`)	Server-side timestamp

14.4 TypeScript ↔ Database Alignment

TypeScript Field	Database Column	Aligned?
`id`	`id`	Yes
`type`	`type`	Yes
`timestamp`	`timestamp`	Yes
`device_id`	`device_id`	Yes
`workspace_id`	`workspace_id`	Yes (canonical on wire, ULID in DB)
`session_id`	`session_id`	Yes
`data`	`data`	Yes
`ingested_at`	`ingested_at`	Yes
`blob_refs`	`blob_refs`	Yes (always `[]`)

The Event interface is the **best-aligned** TypeScript type in the codebase — all 9 fields match their database counterparts exactly.

---

15. The Handler Registry

`packages/core/src/handlers/index.ts`

15.1 Registry Pattern

```
class EventHandlerRegistry {
  private handlers = new Map<EventType, EventHandler>();

  register(eventType: EventType, handler: EventHandler): void {
    this.handlers.set(eventType, handler);  // Last registration wins
  }

  getHandler(eventType: EventType): EventHandler | undefined {
    return this.handlers.get(eventType);
  }

  listRegisteredTypes(): EventType[] {
    return [...this.handlers.keys()];
  }
}
```

One handler per type: If you register twice for the same type, the second replaces the first. This is intentional — allows test doubles to override production handlers.

15.2 Factory Function

```
function createHandlerRegistry(logger?: Logger): EventHandlerRegistry {
  const registry = new EventHandlerRegistry();
  registry.register("session.start", handleSessionStart, logger);
  registry.register("session.end", handleSessionEnd, logger);
  registry.register("git.commit", handleGitCommit, logger);
  registry.register("git.push", handleGitPush, logger);
  registry.register("git.checkout", handleGitCheckout, logger);
  registry.register("git.merge", handleGitMerge, logger);
  return registry;
}
```

6 handlers registered. The remaining 8 event types (`session.compact` , all `remote.*` , all `system.*`) have no handlers — events of these types are persisted to the `events` table but trigger no further processing.

---

16. Individual Handler Deep Dives

16.1 `handleSessionStart` (`packages/core/src/handlers/session-start.ts`)

Trigger: `session.start` event **Output**: Row in `sessions` table with `lifecycle='detected'`

```
1. Extract from event.data:
   - cc_session_id (used as primary key)
   - git_branch, model, source
2. INSERT INTO sessions:
   id = cc_session_id   (NOT a ULID — CC's own UUID)
   workspace_id = ctx.workspaceId (resolved ULID)
   device_id = event.device_id
   lifecycle = 'detected'
   started_at = event.timestamp
   model = event.data.model
   source = event.data.source
   ON CONFLICT (id) DO NOTHING  (idempotent)
3. checkGitHooksPrompt():
   IF workspace != '_unassociated'
   AND git_hooks_installed = false
   AND git_hooks_prompted = false
   THEN SET pending_git_hooks_prompt = true
```

Key insight: The session's `id` is Claude Code's own UUID, not a server-generated ULID. This breaks the convention used by every other entity in the system (which use ULIDs). The reason: CC already has a stable session identifier, and using it as the primary key allows deduplication and direct lookup without a mapping table.

16.2 `handleSessionEnd` (`packages/core/src/handlers/session-end.ts`)

Trigger: `session.end` event **Output**: Session transitions to `lifecycle='ended'`, transcript pipeline triggered

```
1. Extract from event.data:
   - cc_session_id, end_reason, duration_ms
2. If duration_ms == 0 or missing:
   - Compute from started_at → event.timestamp
3. transitionSession():
   UPDATE sessions
   SET lifecycle = 'ended',
       ended_at = event.timestamp,
       end_reason = event.data.end_reason,
       duration_ms = computed_duration
   WHERE id = cc_session_id
     AND lifecycle IN ('detected', 'capturing')
   (optimistic locking — fails if session already ended)
4. If pipelineDeps available AND transcript_s3_key set:
   → pipelineDeps.enqueueSession(sessionId)
   → OR fallback: runSessionPipeline(deps, sessionId) directly
```

The duration_ms=0 pattern: The CLI hook sends `duration_ms: 0` because it doesn't know when the session started. The server computes the actual duration from `sessions.started_at` to `event.timestamp`. If this computation fails (e.g., session not found), it defaults to `0`.

16.3 `handleGitCommit` (`packages/core/src/handlers/git-commit.ts`)

Trigger: `git.commit` event **Output**: Row in `git_activity` table, optional session correlation

```
1. Extract from event.data:
   - hash, message, author_name, author_email
   - branch, files_changed, insertions, deletions, file_list
2. Correlate to session:
   correlateGitEventToSession(sql, workspaceId, deviceId, eventTimestamp)
   → Finds most recent active session on same workspace+device
   → Returns { sessionId, confidence: 'active'|'none' }
3. Transaction:
   a. INSERT INTO git_activity:
      id = event.id (same ULID)
      type = 'commit'
      branch, commit_sha = hash, message
      files_changed, insertions, deletions
      data = { author_name, author_email, file_list }
      session_id = correlated session or NULL
      ON CONFLICT (id) DO NOTHING
   b. IF session correlated:
      UPDATE events SET session_id = sessionId WHERE id = event.id
```

Transaction scope: The `git_activity` INSERT and `events.session_id` UPDATE happen atomically. If either fails, both roll back. This prevents orphaned correlations.

`event.id` reuse: The `git_activity.id` is the **same ULID** as the originating event. This creates a 1:1 link between the event and its materialized view without needing a foreign key. You can always find the source event for a git activity row by looking up `events.id = git_activity.id`.

16.4 `handleGitPush` (`packages/core/src/handlers/git-push.ts`)

Trigger: `git.push` event **Output**: Row in `git_activity` with type='push'

```
1. Extract: branch, remote, commit_count, commits
2. Correlate to session (same as git.commit)
3. Transaction:
   INSERT INTO git_activity:
     type = 'push'
     branch = branch
     commit_sha = NULL, message = NULL
     files_changed = NULL, insertions = NULL, deletions = NULL
     data = { remote, commit_count, commits }
   Update events.session_id if correlated
```

Sparse columns: Push events only populate `branch` and `data`. All other first-class columns (`commit_sha`, `message`, `files_changed`, `insertions`, `deletions`) are NULL.

16.5 `handleGitCheckout` (`packages/core/src/handlers/git-checkout.ts`)

Trigger: `git.checkout` event **Output**: Row in `git_activity` with type='checkout'

```
1. Extract: from_ref, to_ref, from_branch, to_branch
2. branch = to_branch ?? to_ref  (handles detached HEAD)
3. Correlate to session
4. Transaction:
   INSERT INTO git_activity:
     type = 'checkout'
     branch = to_branch ?? to_ref
     data = { from_ref, to_ref, from_branch, to_branch }
   Update events.session_id if correlated
```

16.6 `handleGitMerge` (`packages/core/src/handlers/git-merge.ts`)

Trigger: `git.merge` event **Output**: Row in `git_activity` with type='merge'

```
1. Extract: merge_commit, message, merged_branch, into_branch, files_changed,
had_conflicts
2. Correlate to session
3. Transaction:
   INSERT INTO git_activity:
     type = 'merge'
     branch = into_branch
     commit_sha = merge_commit
     message = message
     files_changed = files_changed
     insertions = NULL, deletions = NULL
     data = { merged_branch, had_conflicts }
   Update events.session_id if correlated
```

---

17. Cross-Cutting Mechanisms

17.1 Deduplication (Three Layers)

Events are deduplicated at three stages:

Layer	Mechanism	When
HTTP response	Per-event `"accepted"`/`"rejected"` status	At ingest
Redis Stream	Redis auto-generated IDs prevent true duplication	At publish
Postgres	`ON CONFLICT (id) DO NOTHING` on `events` INSERT	At persist

The third layer (Postgres) is the authoritative deduplication point. The first two layers are optimistic — an event could be published to Redis twice (pipeline retry) or POSTed twice (queue drain overlap). The `ON CONFLICT` guarantees exactly-once persistence.

17.2 Transport Hints (`_device_name` , `_device_type`)

A pragmatic hack for propagating device metadata without a separate registration endpoint:

```
CLI emit command
  ├─ Injects event.data._device_name = "Johns-MacBook-Pro"
  ├─ Injects event.data._device_type = "local"
  │
  ↓  (travels through HTTP → Redis → consumer as part of event.data)
  │
Event processor
  ├─ Extracts _device_name, _device_type from event.data
  ├─ Passes to resolveOrCreateDevice() as hints
  └─ DELETES _device_name, _device_type from event.data
     (before INSERT INTO events)
```

Properties:

- Untyped — `_device_name` and `_device_type` are not in any TypeScript interface
- Mutates the event object in-place (violates "events are immutable" at the application layer)
- Stripped before persistence (never leaked into the `data` JSONB column)
- Underscore prefix signals "internal, don't rely on this"

17.3 Workspace ID Resolution

The `workspace_id` field undergoes a transformation during processing:

```
On the wire (CLI → HTTP → Redis):
  event.workspace_id = "github.com/user/repo"  (canonical string)

In the processor:
  resolvedWorkspaceId = resolveOrCreateWorkspace(sql, "github.com/user/repo")
  → Returns ULID: "01HXYZ..."
```

```
In Postgres:
  events.workspace_id = "01HXYZ..."  (ULID, FK to workspaces.id)
```

This means the `workspace_id` in the `Event` TypeScript interface is semantically different depending on where in the pipeline you read it:

- Before processing: canonical string
- After processing: ULID

17.4 Session Correlation (Git Events)

Git events arrive without a `session_id` (hooks don't know about CC sessions). The session is correlated server-side:

```
correlateGitEventToSession(sql, workspaceId, deviceId, eventTimestamp)
```

Algorithm: Find the most recently started session on the same workspace+device that is currently active (`lifecycle IN ('detected', 'capturing')`) and `started_at <= eventTimestamp` . Returns `{ sessionId, confidence }` .

If correlated: The `events.session_id` column is **updated** (the only mutation of an event after insert) to link the event to the session.

If not correlated: `session_id` stays NULL. The event becomes an "orphan" — visible in workspace views but not session views.

17.5 `blob_refs` — Declared but Never Populated

The `Event` interface includes `blob_refs: BlobRef[]` for S3 references. The database has a `blob_refs JSONB` column. But:

- The CLI always sends `blob_refs: []`
- No handler populates blob_refs
- Transcript S3 keys are stored on the `sessions` table, not as blob_refs on events
- The `BlobRef` type exists (`{ key, content_type, size_bytes }`) but is never instantiated

This is a forward declaration for a feature that was designed (large payload offloading to S3) but never needed — transcripts ended up going through their own upload path rather than the event pipeline.

---

18. Gap Analysis: What's Missing or Misaligned

Gap 1: 8 of 14 Event Types Are Forward Declarations

Severity: Medium

Only 6 event types have full implementations (hook → emit → handler → output). `session.compact` has a schema but no hook and no handler. The 7 remaining types (`remote.*` , `system.*`) have only their string literals in the `EventType` union — no schemas, no emitters, no handlers.

Impact: The type system allows constructing events with these types, and they would be accepted at the HTTP layer and persisted. But they'd trigger no side effects and be queryable only via raw `SELECT * FROM events WHERE type = '...'` .

Missing Type	What It Would Do	Blocked By
`session.compact`	Upload mid-session transcript backup	No `PreCompact` CC hook
`remote.*` (4 types)	Remote env lifecycle tracking	Phase 5 unimplemented
`system.device.register`	Explicit device registration	Replaced by hint transport
`system.hooks.installed`	Track hook installation per workspace	No emitter
`system.heartbeat`	Periodic device liveness signal	No emitter, no handler

Gap 2: Validated Schemas Discarded at Handler Boundary

Severity: Medium

The Zod payload registry validates `event.data` at ingest time and returns typed results (e.g., `SessionStartPayload`). But handlers don't use the validated/typed result. Instead:

```
// Handler casts from untyped event.data
const ccSessionId = event.data.cc_session_id as string;
const gitBranch = event.data.git_branch as string | null;
```

The `as string` casts give false type safety — they're assertions, not validations. If a field is missing or the wrong type, TypeScript won't catch it.

What should happen: The handler should receive the validated, typed payload from the registry, eliminating the need for unsafe casts.

Gap 3: `session.compact` Has Schema But No Pipeline

Severity: Medium

CORE.md specifies a `PreCompact` hook that fires when CC compacts its context window. The schema exists (`session-compact.ts`), the type exists in the union, but:

- No CC hook emits `session.compact` events
- No handler processes them
- The schema's `compact_sequence` field mirrors the transcript parser's unused `compact_sequence` column

This means context window compactions — a significant event in long CC sessions — are invisible to fuel-code unless the transcript is parsed after the session ends.

Gap 4: `events.session_id` Mutation After Insert

Severity: Low-Medium

CORE.md states events are "never updated or deleted." But git event handlers UPDATE `events.session_id` after insert to record the session correlation:

```
UPDATE events SET session_id = $sessionId WHERE id = $eventId
```

This is the only mutation of an event row in the entire system. It's pragmatically necessary (the session_id isn't known at emission time for git events) but technically violates the immutability invariant.

Alternative: Store the correlation in `git_activity.session_id` only, and join when needed. But this would make `events.session_id` useless for git events and break the partial index.

Gap 5: No Event Query API

Severity: Medium

There is no `GET /api/events` endpoint. Events are not directly queryable — they're only visible indirectly through:

- Session detail (events linked to a session)
- Timeline (events displayed as session items or orphan git groups)
- Git activity (materialized from git events)

If you want to see all `session.compact` events, all events from a specific device in a time range, or raw event data by ID — there's no API for it. You'd need to query Postgres directly.

Gap 6: `blob_refs` Never Used

Severity: Low

The `Event.blob_refs` field and `events.blob_refs` column exist but are always `[]`. The `BlobRef` interface is defined but never instantiated. The S3 blob offloading feature described in CORE.md was designed into the event schema but superseded by the transcript upload path.

Impact: Dead weight in the schema, type, and every serialization/deserialization path.

Gap 7: Clock Skew Between Client and Server

Severity: Low-Medium

Events have two timestamps:

- `timestamp` — from the originating machine's clock (set by CLI)
- `ingested_at` — from the server's clock (set by event processor)

If the client clock is skewed (common on laptops after sleep, VMs with poor NTP), `timestamp` could be in the future or past relative to `ingested_at`. The session correlator uses `event.timestamp` for matching (`started_at <= eventTimestamp`), which means a skewed clock could cause incorrect session correlation.

No clock skew detection or correction exists.

Gap 8: No Event Retention / TTL

Severity: Low

CORE.md mentions TTL-based data retention but no retention mechanism exists:

- No background job prunes old events
- No `expired_at` or `retain_until` column
- The `events` table grows indefinitely

For a single-user system this is currently fine, but it means the table will grow without bound.

Gap 9: `ingestRequestSchema` Allows Batch But CLI Always Sends 1

Severity: Low

The ingest endpoint accepts batches of 1-100 events. But the CLI `emit` command always sends exactly one event per request. The local queue drain sends batches (up to `batch_size: 50`), but normal real-time emission is unbatched. This means every hook invocation creates a separate HTTP request — 4 hooks firing in quick succession means 4 HTTP requests.

Impact: Slightly higher latency and connection overhead. Not problematic at single-user scale.

Gap 10: No Backpressure Between Consumer and Processor

Severity: Low

The consumer reads 10 events at a time and processes them sequentially. If processing is slow (e.g., due to slow Postgres), the consumer just slows down — it reads fewer events per time period, and unread events accumulate in the Redis Stream.

There's no:

- Adaptive batch sizing based on processing time
- Circuit breaker on the processor
- Metrics on consumer lag relative to publisher rate

At single-user scale (tens of events per minute), this is not a concern.

Summary Table

Gap	Severity	Impact	Fix Complexity
8 of 14 types are stubs	Medium	Forward declarations without implementation	High (requires hooks, handlers)
Schemas discarded at handler boundary	Medium	Unsafe `as` casts, no type safety in handlers	Low (pass validated types)
`session.compact` incomplete	Medium	Compaction events invisible	Medium (needs CC hook + handler)
`events.session_id` mutation	Low-Med	Violates immutability invariant	Low (accept or restructure)
No event query API	Medium	Raw events not directly accessible	Low (add GET endpoint)
`blob_refs` never used	Low	Dead schema weight	Low (remove or document as planned)
No clock skew handling	Low-Med	Possible mis-correlation	Medium (detect, warn, or use server time)
No event retention/TTL	Low	Unbounded table growth	Medium (background prune job)

Unbatched real-time emit	Low	Extra HTTP overhead	Low (batch window in CLI)
No consumer backpressure	Low	Irrelevant at current scale	Medium (adaptive batching)

---

19. References

#	File	Description
1	`packages/shared/src/types/event.ts`	Core `Event` interface, `EventType` union, `BlobRef`, `IngestRequest/Response`
2	`packages/shared/src/schemas/event-base.ts`	Zod envelope schema: `eventSchema`, `ingestRequestSchema`, ULID regex
3	`packages/shared/src/schemas/session-start.ts`	`session.start` payload schema
4	`packages/shared/src/schemas/session-end.ts`	`session.end` payload schema
5	`packages/shared/src/schemas/session-compact.ts`	`session.compact` payload schema
6	`packages/shared/src/schemas/git-commit.ts`	`git.commit` payload schema
7	`packages/shared/src/schemas/git-push.ts`	`git.push` payload schema
8	`packages/shared/src/schemas/git-checkout.ts`	`git.checkout` payload schema
9	`packages/shared/src/schemas/git-merge.ts`	`git.merge` payload schema
10	`packages/shared/src/schemas/payload-registry.ts`	`PAYLOAD_SCHEMAS` map, `validateEventPayload()`
11	`packages/shared/src/schemas/index.ts`	Barrel export for all schemas
12	`packages/shared/src/types/ws.ts`	WebSocket protocol types (subscribe, event, session.update, etc.)
13	`packages/core/src/event-processor.ts`	`processEvent()`, `EventHandlerRegistry`, `EventHandlerContext`
14	`packages/core/src/handlers/index.ts`	`createHandlerRegistry()` — registers all 6 handlers
15	`packages/core/src/handlers/session-start.ts`	`handleSessionStart` — creates session record
16	`packages/core/src/handlers/session-end.ts`	`handleSessionEnd` — transitions session, triggers pipeline
17	`packages/core/src/handlers/git-commit.ts`	`handleGitCommit` — materializes git_activity row
18	`packages/core/src/handlers/git-push.ts`	`handleGitPush` — materializes git_activity row

19	`packages/core/src/handlers/git-checkout.ts`	`handleGitCheckout` — materializes git_activity row
20	`packages/core/src/handlers/git-merge.ts`	`handleGitMerge` — materializes git_activity row
21	`packages/core/src/git-correlator.ts`	`correlateGitEventToSession()` — session matching heuristic
22	`packages/core/src/session-lifecycle.ts`	Session state machine, `transitionSession()` with optimistic locking
23	`packages/server/src/routes/events.ts`	`POST /api/events/ingest` — HTTP ingest endpoint
24	`packages/server/src/redis/stream.ts`	Redis Stream operations: publish, read, ack, claim
25	`packages/server/src/pipeline/consumer.ts`	Consumer main loop: read → process → ack/retry
26	`packages/server/src/pipeline/wire.ts`	`createEventHandler()` — wires registry to SQL client
27	`packages/server/src/ws/broadcaster.ts`	`WsBroadcaster` — fan-out to subscribed WebSocket clients
28	`packages/server/src/ws/index.ts`	WebSocket server: auth, keepalive, subscription management
29	`packages/server/src/ws/types.ts`	`ConnectedClient` interface
30	`packages/server/src/db/migrations/001_initial.sql`	`events` table schema + 4 indexes
31	`packages/cli/src/commands/emit.ts`	`fuel-code emit` — event packaging, POST, queue fallback
32	`packages/cli/src/lib/queue.ts`	Local queue: `enqueueEvent()`, `listQueuedEvents()`, `moveToDeadLetter()`
33	`packages/cli/src/lib/drain.ts`	`drainQueue()` — batch retry with dead-letter after 100 attempts
34	`packages/cli/src/lib/drain-background.ts`	`spawnBackgroundDrain()` — detached process with lockfile
35	`packages/cli/src/lib/ws-client.ts`	`WsClient` — WebSocket client with auto-reconnect
36	`packages/cli/src/lib/api-client.ts`	`createApiClient()`, `FuelApiClient` — HTTP client for ingest
37	`packages/hooks/claude/SessionStart.sh`	CC hook: session start wrapper
38	`packages/hooks/claude/_helpers/session-start.ts`	CC hook: session start logic

39	`packages/hooks/claude/SessionEnd.sh`	CC hook: session end wrapper
40	`packages/hooks/claude/_helpers/session-end.ts`	CC hook: session end logic
41	`packages/hooks/claude/_helpers/resolve-workspace.ts`	TS workspace resolution for CC hooks
42	`packages/hooks/git/post-commit`	Git hook: commit metadata extraction
43	`packages/hooks/git/pre-push`	Git hook: push ref parsing, local hook delegation
44	`packages/hooks/git/post-checkout`	Git hook: branch switch detection
45	`packages/hooks/git/post-merge`	Git hook: merge metadata extraction
46	`packages/hooks/git/resolve-workspace.sh`	Bash workspace resolution for git hooks
47	`tasks/CORE.md`	Definitive system spec — Event abstraction definition (Section 3)