

fuel-code: Full-Stack Processing Flows

Comprehensive technical reference documenting every data flow in the system — from event generation through processing to final persistence. Intended as the definitive guide for anyone making architectural decisions in this codebase.

Last updated: 2026-02-27 **Codebase state:** main @ 6cfc158

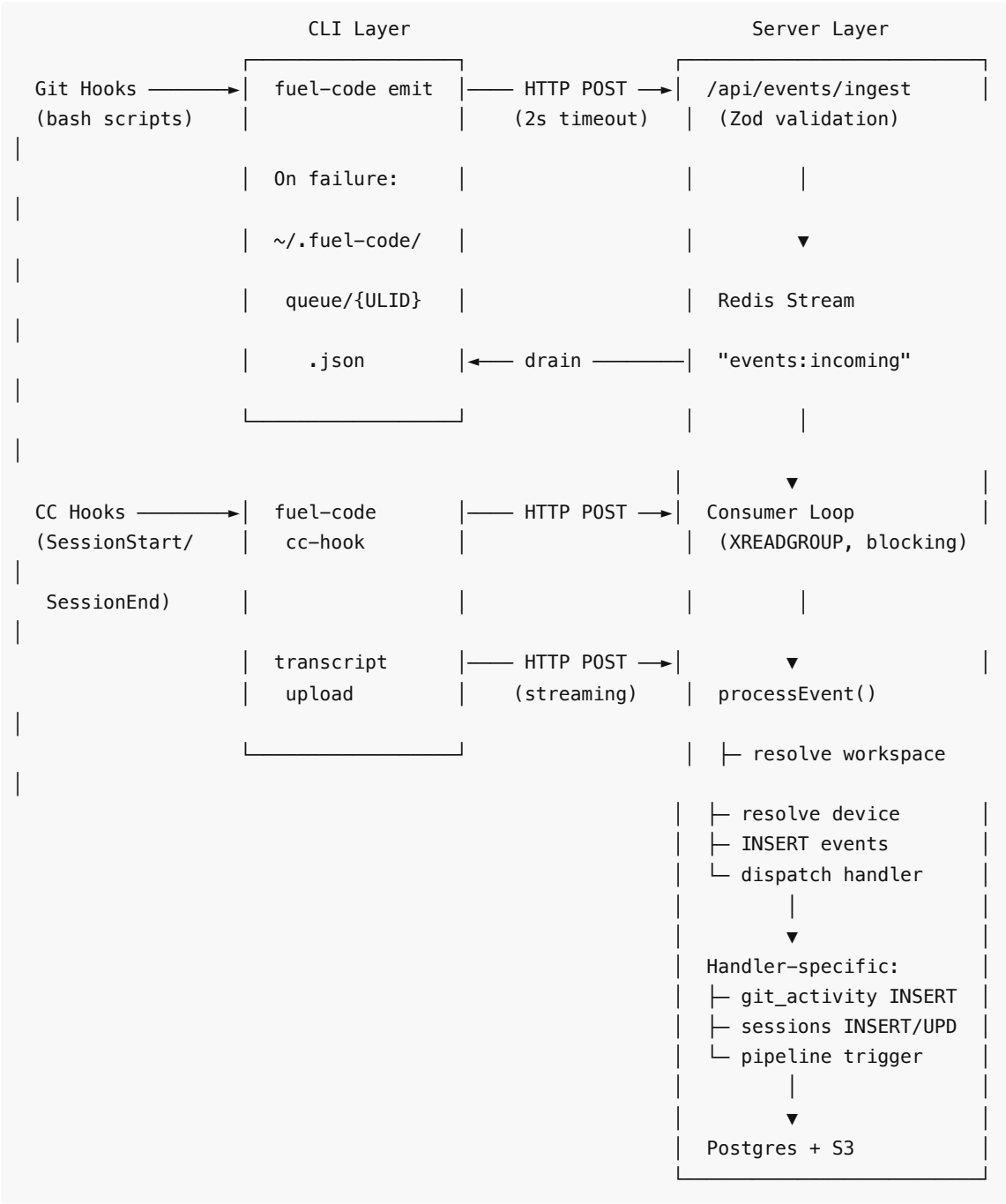
Table of Contents

1. [System Overview](#)
 2. [Git Activity Flow](#)
 - 2.1 Hook Installation
 - 2.2 Event Generation (Shell Scripts)
 - 2.3 Workspace Resolution
 - 2.4 CLI Emit Path
 - 2.5 Local Queue Fallback
 - 2.6 Queue Drain
 3. [Server-Side Event Ingestion](#)
 - 3.1 HTTP Ingest Endpoint
 - 3.2 Redis Streams Transport
 - 3.3 Consumer Loop
 - 3.4 Event Processor (Core)
 - 3.5 Git Event Handlers
 - 3.6 Git-Session Correlation
 4. [Transcript Flow](#)
 - 4.1 Claude Code Hook Capture
 - 4.2 Transcript Upload to S3
 - 4.3 Post-Processing Pipeline
 - 4.4 JSONL Transcript Parser
 - 4.5 Persistence (Messages + Content Blocks)
 - 4.6 LLM Summary Generation
 - 4.7 Pipeline Queue & Concurrency
 5. [Session Lifecycle](#)
 - 5.1 State Machine
 - 5.2 Optimistic Locking Transitions
 - 5.3 Recovery & Reparse
 6. [Real-Time Broadcasting](#)
 7. [Database Schema Reference](#)
 8. [Error Handling & Resilience](#)
 9. [Future Architecture](#)
 - 9.1 Phase 5: Remote Dev Environments
 - 9.2 V2: Analysis Layer
-

1. System Overview

fuel-code is a CLI-first developer activity tracking system. It captures two primary data streams — **git activity** and **Claude Code session transcripts** — and processes them through a unified event pipeline.

Architecture at a Glance



Package Structure

Package	Purpose	I/O Knowledge
packages/shared/	Types, schemas, utilities	None (pure)
packages/core/	Domain logic: processors, handlers, parsers	None (injected deps)
packages/server/	Express API, Redis, S3, WebSocket	Full (owns infrastructure)

packages/cli/	CLI commands, TUI, API client	HTTP client only
packages/hooks/	Git + CC hook scripts	Shell only

Critical design invariant: `core/` has zero knowledge of HTTP, WebSocket, CLI, or any UI framework. All infrastructure is injected via interfaces. This enables the "UI swappability" principle — any consumer (CLI, web, future mobile) talks to the same server API.

2. Git Activity Flow

2.1 Hook Installation

Two installation modes exist for git hooks:

Global mode (default): Installs to `~/.fuel-code/git-hooks/` and sets `git config --global core.hooksPath .`

Per-repo mode (`--per-repo`): Installs directly into `.git/hooks/` of the current repository.

```
// packages/cli/src/lib/git-hook-installer.ts:156-247
// Installation steps:
// 1. Verify git is available and ~/.fuel-code/ exists
// 2. Detect competing hook managers (Husky, Lefthook, pre-commit) – abort unless --force
// 3. Backup existing hooks as <hook>.user for chaining
// 4. Copy scripts from packages/hooks/git/
// 5. chmod +x all scripts
// 6. Set core.hooksPath (global mode)
```

Four git hooks are installed (defined at `packages/cli/src/lib/git-hook-installer.ts:26-31`):

- `post-commit` — fires after commit creation
- `post-checkout` — fires after branch switch
- `post-merge` — fires after merge completion
- `pre-push` — fires before push (can block push if user hook exits non-zero)

Claude Code hooks are also installed into `~/.claude/settings.json`:

```
// packages/cli/src/commands/hooks.ts:180-263
// CC hooks use bash backgrounding to prevent blocking Claude Code:
// bash -c 'data=$(cat); printf "%s" "$data" | fuel-code cc-hook session-start &'
// Two hooks: SessionStart -> fuel-code cc-hook session-start
//             SessionEnd   -> fuel-code cc-hook session-end
```

Hook chaining: Existing user hooks are preserved as `<hook>.user` files. `fuel-code` hooks invoke the `.user` hook first to maintain other integrations.

2.2 Event Generation (Shell Scripts)

Each git hook follows the same pattern: **dispatch to user hook** -> **resolve workspace** -> **extract metadata** -> **emit event** -> **exit 0** always.

post-commit (packages/hooks/git/post-commit)

```
# packages/hooks/git/post-commit – key sections:

# Safety: always exits 0, never blocks git (line 99)
# Chains to .user hook if present (lines 15-19)
# Per-repo opt-out: .fuel-code/config.yaml with git_enabled: false (lines 27-33)

# Metadata extraction (lines 42-76):
COMMIT_HASH=$(git rev-parse HEAD)
COMMIT_MSG=$(git log -1 --pretty=%B HEAD | head -c 8192)
AUTHOR_NAME=$(git log -1 --pretty=%an HEAD)
AUTHOR_EMAIL=$(git log -1 --pretty=%ae HEAD)
BRANCH=$(git rev-parse --abbrev-ref HEAD)

# Diff stats (lines 54-60) – skips binary files marked with "-":
git diff-tree --numstat -r HEAD | while read adds dels file; do
    [ "$adds" != "-" ] && INSERTIONS=$((INSERTIONS + adds))
    [ "$dels" != "-" ] && DELETIONS=$((DELETIONS + dels))
done

# File list: JSON array via python3 json.dumps() (lines 62-76)

# Emission (lines 78-97) – fire-and-forget in background:
fuel-code emit git.commit \
    --workspace-id "$WORKSPACE_ID" \
    --data-stdin <<HEREDOC
{ "hash": "$COMMIT_HASH", "message": "$COMMIT_MSG", ... }
HEREDOC
```

Every hook runs `fuel-code emit` in the background (`&`) with errors piped to `~/.fuel-code/hook-errors.log` . This ensures the git operation itself is never delayed.

pre-push (packages/hooks/git/pre-push)

Pre-push has unique constraints:

- **MUST read stdin** (git expects it consumed)
- **MUST exit 0** to avoid blocking the push
- Respects user hook exit codes (can legitimately block push)
- Reads stdin format: `ref local_sha remote_ref remote_sha` per line
- Skips branch deletions (all-zero local_sha) and tag pushes
- Emits `git.push` with: `branch, remote, commit_count, commits[]`

post-checkout (packages/hooks/git/post-checkout)

- Only fires on branch switch (`$3=1`), not file checkout
- Handles detached HEAD (sets `to_branch` to null)
- Emits `git.checkout` with: `from_branch, to_branch, from_ref, to_ref`

post-merge (packages/hooks/git/post-merge)

- Emits `git.merge` with: `merge_commit, into_branch, files_changed, had_conflicts`

2.3 Workspace Resolution

Every hook calls `resolve-workspace.sh` to produce a **canonical workspace ID** — a deterministic string computed entirely client-side, with no server round-trip.

```
# packages/hooks/git/resolve-workspace.sh:1-61

# Resolution priority:
# 1. git remote get-url origin
# 2. First remote alphabetically (if no "origin")
# 3. local:<sha256(first_commit_hash)> for local-only repos
# 4. Exit 1 if no remote and no commits

# Normalization:
# SSH:   git@github.com:user/repo.git → github.com/user/repo
# HTTPS: https://github.com/user/repo.git → github.com/user/repo
# Always: lowercase host, strip .git suffix
```

The same normalization logic exists in TypeScript (`packages/shared/src/canonical.ts`) for the server side, ensuring both CLI and server produce identical canonical IDs for the same repository.

2.4 CLI Emit Path

The `emit` command is the most performance-critical path — it must complete within ~2 seconds and always exit 0.

```
// packages/cli/src/commands/emit.ts:90-194

export async function runEmit(eventType: string, options: EmitOptions):
Promise<void> {
  // 1. Load config — if missing, queue with hardcoded path
  const config = loadConfig();

  // 2. Parse event data from stdin (--data-stdin) or --data argument
  //   Invalid JSON wrapped as { _raw: theString }
  let data = parseData(options);

  // 3. Inject device hints for backend registration (lines 146-149)
  data._device_name = config.device.name;
  data._device_type = config.device.type;

  // 4. Construct Event with ULID + timestamps (lines 151-161)
  const event: Event = {
    id: generateId(),           // ULID — time-sortable, globally unique
    type: eventType,
    timestamp: new Date().toISOString(),
    device_id: config.device.id,
    workspace_id: options.workspaceId ?? "_unassociated",
    session_id: options.sessionId ?? null,
    data,
    ingested_at: null,
  }
```

```

    blob_refs: [],
  };

  // 5. Attempt HTTP POST to /api/events/ingest (2-second timeout)
  try {
    await client.ingest([event]);
    return; // Success – exit 0 silently
  } catch {
    // 6. On ANY failure – fall through to local queue
  }

  // 7. Queue fallback – NEVER throws
  enqueueEvent(event, config.pipeline.queue_path);
}

```

Key properties:

- Always exits 0 (hooks must never fail)
- Zero stdout on success (hooks are silent)
- 2-second HTTP timeout (configurable via `config.pipeline.post_timeout_ms`)
- Device hints (`_device_name` , `_device_type`) are stripped server-side before persistence

2.5 Local Queue Fallback

When the backend is unreachable, events are persisted to the local filesystem.

```

// packages/cli/src/lib/queue.ts:52-82

export function enqueueEvent(event: Event, queueDir: string): string | null {
  // Atomic write: write to .tmp file, then rename (POSIX atomic)
  const filename = `${event.id}.json`; // ULID = chronological sort order
  const tmpPath = path.join(queueDir, `.${filename}.tmp.${randomHex()}`);
  const finalPath = path.join(queueDir, filename);

  fs.writeFileSync(tmpPath, JSON.stringify(event), { mode: 0o600 });
  fs.renameSync(tmpPath, finalPath); // Atomic on POSIX

  return finalPath;
  // NEVER throws – last-resort fallback
}

```

Queue directory: `~/.fuel-code/queue/` **Dead-letter directory:** `~/.fuel-code/dead-letter/`

Filenames: ULID-based, so `ls | sort` = chronological order

2.6 Queue Drain

Queued events are drained by the `drain` module, which can run in foreground (`fuel-code queue drain`) or be spawned in the background after a queue write.

```

// packages/cli/src/lib/drain.ts:104-249

// Drain flow:

```

```
// 1. List all queued events (ULID-sorted)
// 2. Batch into groups of config.pipeline.batch_size (default 50)
// 3. For each batch:
//   - Read each event file, move corrupted files to dead-letter
//   - Check _attempts counter, dead-letter if >= 100
//   - POST batch to /api/events/ingest (10-second timeout)
//   - On success: remove event files
//   - On 401 (invalid API key): stop immediately
//   - On 503/timeout: stop, increment attempts
//   - On other network error: stop, increment attempts
```

Attempt tracking: Each event file stores an `_attempts` counter. On failure the counter is incremented and the file is rewritten atomically. Events exceeding 100 attempts are moved to `~/.fuel-code/dead-letter/`.

3. Server-Side Event Ingestion

3.1 HTTP Ingest Endpoint

```
// packages/server/src/routes/events.ts:59-168
// POST /api/events/ingest

// Step 1: Validate envelope (Zod) – rejects ENTIRE batch on failure (HTTP 400)
parsed = ingestRequestSchema.parse(req.body);
// Validates: ULIDs match /^[0-9A-HJKMNP-TV-Z]{26}$/, batch size 1-100, required
fields

// Step 2: Validate type-specific payloads per event
for (const event of parsed.events) {
  const payloadResult = validateEventPayload(event.type, event.data);
  if (!payloadResult.success) {
    // Reject THIS event, not the batch – forward-compatible for unknown types
    results.push({ index: i, status: "rejected" });
    continue;
  }
  // Set server-side ingestion timestamp
  const fullEvent = { ...event, ingested_at: new Date().toISOString() };
  valid.push(fullEvent);
}

// Step 3: Publish valid events to Redis Stream
const publishResult = await publishBatchToStream(redis, valid);
// Per-event publish failures marked as rejected

// Step 4: Return 202 Accepted with per-event results
res.status(202).json({
  ingested: publishedCount,
  duplicates: 0,
  rejected: rejectedCount,
  results, // [{index, status: "accepted"|"rejected"}]
```

```

    errors,          // [{index, error: "..."}]
  });

```

Design decisions:

- **202 Accepted** (not 200 OK) because processing is asynchronous via Redis
- Envelope validation rejects the entire batch; payload validation rejects individual events
- Unregistered event types pass through (forward-compatible)
- Total Redis failure returns 503 with `retry_after_seconds: 30`

3.2 Redis Streams Transport

```

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

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

// Serialization (lines 101-113):
// Events are flattened to Redis Stream key-value pairs:
function serializeEvent(event: Event): 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
    "ingested_at", event.ingested_at ?? "",
    "blob_refs", JSON.stringify(event.blob_refs), // JSON-encoded
  ];
}

// Consumer group setup (lines 75-89):
await redis.xgroup("CREATE", EVENTS_STREAM, CONSUMER_GROUP, "0", "MKSTREAM");
// MKSTREAM creates the stream if it doesn't exist
// "0" means the group starts reading from the beginning
// BUSYGROUP error silently ignored (group already exists)

// Batch publishing (lines 180-224):
// Uses Redis pipeline for single round-trip (all XADDs sent at once)
const pipeline = redis.pipeline();
for (const event of events) {
  pipeline.xadd(EVENTS_STREAM, "*", ...serializeEvent(event));
}
const results = await pipeline.exec();

// Consumer reading (lines 263-294):
const response = await redis.xreadgroup(
  "GROUP", CONSUMER_GROUP, CONSUMER_NAME,

```

```

    "COUNT", count,          // max 10 entries per read
    "BLOCK", blockMs,         // 5000ms – keeps CPU idle when quiet
    "STREAMS", EVENTS_STREAM, ">" // ">" = new messages only
  );

  // Acknowledgement (lines 310–324):
  await redis.xack(EVENTS_STREAM, CONSUMER_GROUP, streamId);
  // Removes entry from Pending Entries List (PEL)

  // Pending entry reclamation (lines 345–451):
  // On startup, reclaims entries idle >60s from crashed consumers
  // Tries XAUTOCLAIM first (Redis 6.2+), falls back to XPENDING+XCLAIM

```

3.3 Consumer Loop

The consumer is a single long-running loop that reads from the Redis Stream and dispatches each event to the processor.

```

// packages/server/src/pipeline/consumer.ts:125–368

// Constants:
const READ_COUNT = 10;           // Max entries per iteration
const BLOCK_MS = 5_000;          // 5s blocking read
const CLAIM_IDLE_MS = 60_000;    // Reclaim after 60s idle
const CLAIM_COUNT = 100;         // Max pending to reclaim on startup
const MAX_RETRIES = 3;           // Max failures before dead-letter
const DEFAULT_RECONNECT_DELAY_MS = 5_000; // Retry delay on Redis errors
const STOP_TIMEOUT_MS = 10_000;  // Max wait for graceful shutdown

export function startConsumer(deps: ConsumerDeps): ConsumerHandle {
  // In-memory retry counter: streamId -> failure count
  const failureCounts = new Map<string, number>();

  async function loop(): Promise<void> {
    // --- Startup: ensure consumer group (retry until success) ---
    while (!shouldStop) {
      try { await ensureGroup(redis); break; }
      catch { await sleep(reconnectDelayMs); }
    }

    // --- Reclaim pending entries from crashed consumers ---
    const pending = await claimPending(redis, CLAIM_IDLE_MS, CLAIM_COUNT);
    for (const entry of pending) await handleEntry(entry);

    // --- Main loop ---
    while (!shouldStop) {
      try {
        const entries = await readStream(redis, READ_COUNT, BLOCK_MS);
        for (const entry of entries) await handleEntry(entry);
      } catch (err) {
        // Detect NOGROUP (Redis restarted) – re-create group immediately

```

```

        if (errMsg.includes("NOGROUP")) {
            await ensureGroup(redis);
            continue; // Skip delay
        }
        await sleep(reconnectDelayMs);
    }
}

async function handleEntry(entry: StreamEntry): Promise<void> {
    try {
        const result = await processEvent(sql, entry.event, registry, logger,
pipelineDeps);

        // Broadcast to WebSocket clients on success
        if (broadcaster && result.status !== "duplicate") {
            broadcaster.broadcastEvent(entry.event);
            // Also broadcast session lifecycle transitions
            if (entry.event.type === "session.start") {
                broadcaster.broadcastSessionUpdate(session_id, workspace_id, "detected");
            }
        }

        await ackEntry(redis, entry.streamId);
        failureCounts.delete(entry.streamId);
    } catch {
        const failures = (failureCounts.get(entry.streamId) ?? 0) + 1;
        failureCounts.set(entry.streamId, failures);

        if (failures >= MAX_RETRIES) {
            // Dead-letter: acknowledge to prevent infinite retry
            await ackEntry(redis, entry.streamId);
            failureCounts.delete(entry.streamId);
        }
        // else: leave un-acked for retry on next iteration
    }
}
}

```

Key properties:

- Never crashes the process — all errors caught and retried
- NOGROUP errors (Redis restart) detected and group re-created immediately
- Dead-lettering after 3 failures (ack + error log)
- Graceful shutdown: `stop()` waits for current iteration + 10s timeout

3.4 Event Processor (Core)

The processor is the central function called by the consumer for every event. It lives in `core/` and has no infrastructure knowledge.

```
// packages/core/src/event-processor.ts:132-235
```

```
export async function processEvent(
  sql: Sql,
  event: Event,
  registry: EventHandlerRegistry,
  logger: Logger,
  pipelineDeps?: PipelineDeps,
): Promise<ProcessResult> {

  // 1. Resolve workspace: canonical ID string -> ULID
  //    e.g. "github.com/user/repo" -> "01HXY..." (creates row if new)
  const resolvedWorkspaceId = await resolveOrCreateWorkspace(
    sql, event.workspace_id, extractHints(event),
  );

  // 2. Resolve device: ensure device row exists
  //    Uses _device_name/_device_type hints from CLI
  const deviceHints = event.data._device_name
    ? { name: event.data._device_name, type: event.data._device_type ?? "local" }
    : undefined;
  await resolveOrCreateDevice(sql, event.device_id, deviceHints);

  // 3. Link workspace to device with local path
  await ensureWorkspaceDeviceLink(sql, resolvedWorkspaceId, event.device_id,
    localPath);

  // 4. Strip internal transport hints before persistence
  delete event.data._device_name;
  delete event.data._device_type;

  // 5. Insert event row - ON CONFLICT (id) DO NOTHING deduplicates by ULID
  const insertResult = await sql`
    INSERT INTO events (id, type, timestamp, device_id, workspace_id,
                        session_id, data, blob_refs, ingested_at)
    VALUES (${event.id}, ${event.type}, ${event.timestamp},
            ${event.device_id}, ${resolvedWorkspaceId},
            ${event.session_id}, ${JSON.stringify(event.data)},
            ${JSON.stringify(event.blob_refs)}, ${new Date().toISOString()})
    ON CONFLICT (id) DO NOTHING
    RETURNING id
  `;

  if (insertResult.length === 0) {
    return { eventId: event.id, status: "duplicate", handlerResults: [] };
  }

  // 6. Validate payload against registered schema
  const validation = validateEventPayload(event.type, event.data);
  if (!validation.success) {
    return { eventId: event.id, status: "processed",
```

```

        handlerResults: [{ type: event.type, success: false, error: "Payload
validation failed" }] });
    }

    // 7. Dispatch to type-specific handler
    const handler = registry.getHandler(event.type);
    if (handler) {
        await handler({ sql, event, workspaceId: resolvedWorkspaceId, logger,
pipelineDeps });
    }

    return { eventId: event.id, status: "processed", handlerResults };
}

```

Handler registration happens in `packages/server/src/pipeline/wire.ts`, which maps event types to handler functions:

- `session.start` -> `handleSessionStart`
- `session.end` -> `handleSessionEnd`
- `git.commit` -> `handleGitCommit`
- `git.push` -> `handleGitPush`
- `git.checkout` -> `handleGitCheckout`
- `git.merge` -> `handleGitMerge`

3.5 Git Event Handlers

`handleGitCommit`

```

// packages/core/src/handlers/git-commit.ts:26-88

export async function handleGitCommit(ctx: EventHandlerContext): Promise<void> {
    const { sql, event, workspaceId, logger } = ctx;

    // Extract fields from git.commit payload
    const hash = event.data.hash as string;
    const message = event.data.message as string;
    const branch = event.data.branch as string;
    const filesChanged = event.data.files_changed as number;
    const insertions = event.data.insertions as number;
    const deletions = event.data.deletions as number;
    const fileList = event.data.file_list ?? null;

    // Correlate to active CC session (see section 3.6)
    const correlation = await correlateGitEventToSession(
        sql, workspaceId, event.device_id, new Date(event.timestamp),
    );

    // Transaction: INSERT git_activity + UPDATE events.session_id
    await sql.begin(async (tx) => {
        await tx`
            INSERT INTO git_activity (id, workspace_id, device_id, session_id, type,
                                    branch, commit_sha, message, files_changed,

```

```

        insertions, deletions, timestamp, data)
VALUES (${event.id}, ${workspaceId}, ${event.device_id},
        ${correlation.sessionId}, ${"commit"}, ${branch}, ${hash},
        ${message}, ${filesChanged}, ${insertions}, ${deletions},
        ${event.timestamp},
        ${JSON.stringify({ author_name: authorName, author_email: authorEmail,
file_list: fileList })})
    ON CONFLICT (id) DO NOTHING
`;

// If correlated to a session, backfill the event's session_id
if (correlation.sessionId) {
    await tx`
        UPDATE events SET session_id = ${correlation.sessionId}
        WHERE id = ${event.id} AND session_id IS NULL
    `;
}
});
}

```

Transaction wrapping is critical: the `git_activity` INSERT and `events` UPDATE must both succeed or both roll back, preventing inconsistent state.

The other git handlers (`git-push.ts` , `git-checkout.ts` , `git-merge.ts`) follow the same pattern: extract fields, correlate, transactional insert + session_id backfill.

3.6 Git-Session Correlation

```

// packages/core/src/git-correlator.ts:47-68

export async function correlateGitEventToSession(
    sql: Sql,
    workspaceId: string,
    deviceId: string,
    eventTimestamp: Date,
): Promise<CorrelationResult> {
    // Find the most recently started session that:
    // - Is on the same workspace + device
    // - Is currently active (detected or capturing)
    // - Started before the git event occurred
    const rows = await sql`
        SELECT id FROM sessions
        WHERE workspace_id = ${workspaceId}
            AND device_id = ${deviceId}
            AND lifecycle IN ('detected', 'capturing')
            AND started_at <= ${eventTimestamp.toISOString()}
        ORDER BY started_at DESC
        LIMIT 1
    `;

    if (rows.length > 0) {

```

```

    return { sessionId: rows[0].id, confidence: "active" };
  }
  return { sessionId: null, confidence: "none" };
}

```

When no active session is found, git events are recorded as "orphan" workspace-level activity (`session_id = NULL` in `git_activity`). The timeline endpoint includes these orphan events in its results.

4. Transcript Flow

4.1 Claude Code Hook Capture

Claude Code fires two hooks during a session lifecycle. Both are registered in `~/.claude/settings.json` by `fuel-code hooks install`.

SessionStart → `fuel-code cc-hook session-start` :

```

// packages/cli/src/commands/cc-hook.ts (session-start handling)

// Claude Code provides JSON on stdin:
// { session_id, cwd, transcript_path, source, model }

// The handler:
// 1. Reads JSON from stdin
// 2. Resolves workspace identity from cwd (git remote -> canonical ID)
// 3. Resolves git branch (git symbolic-ref --short HEAD)
// 4. Emits session.start event with:
//    cc_session_id, cwd, git_branch, git_remote, cc_version, model, source,
//    transcript_path

```

SessionEnd → `fuel-code cc-hook session-end` :

```

// packages/cli/src/commands/cc-hook.ts (session-end handling)

// Claude Code provides JSON on stdin:
// { session_id, transcript_path, duration_ms, end_reason }

// The handler:
// 1. Reads JSON from stdin
// 2. Emits session.end event (cc_session_id, duration_ms, end_reason,
//    transcript_path)
// 3. Uploads transcript to S3 via: fuel-code transcript upload <session_id> <path>

```

Backgrounding: CC hooks use bash backgrounding so they never block Claude Code:

```

bash -c 'data=$(cat); printf "%s" "$data" | fuel-code cc-hook session-end &'

```

Stdin is read synchronously *before* backgrounding to avoid data races.

4.2 Transcript Upload to S3

```
// packages/server/src/routes/transcript-upload.ts:78-183
// POST /api/sessions/:id/transcript/upload

// Step 1: Validate Content-Length (max 200MB)
if (contentType > MAX_UPLOAD_BYTES) { // MAX_UPLOAD_BYTES = 200 * 1024 * 1024
  return res.status(413);
}

// Step 2: Validate session exists
const session = await sql`SELECT id, lifecycle, workspace_id, transcript_s3_key FROM
sessions WHERE id = ${sessionId}`;

// Step 3: Idempotency – if transcript already uploaded, return 200
if (session.transcript_s3_key) {
  return res.status(200).json({ status: "already_uploaded", s3_key:
session.transcript_s3_key });
}

// Step 4: Build S3 key from workspace canonical ID
const s3Key = buildTranscriptKey(canonicalId, sessionId);
// Format: transcripts/{canonicalId}/{sessionId}/raw.jsonl

// Step 5: Buffer request body then upload to S3
// Buffering decouples client->server and server->S3 connections so a client
// disconnect mid-stream can't corrupt the S3 upload
const chunks: Buffer[] = [];
for await (const chunk of req) { chunks.push(Buffer.from(chunk)); }
const body = Buffer.concat(chunks);
await s3.upload(s3Key, body, "application/x-ndjson");

// Step 6: Update session with S3 key
await sql`UPDATE sessions SET transcript_s3_key = ${s3Key} WHERE id = ${sessionId}`;

// Step 7: Trigger pipeline if session has already ended
if (session.lifecycle === "ended") {
  triggerPipeline(pipelineDeps, sessionId, logger);
}

// Step 8: Return 202 Accepted
res.status(202).json({ status: "uploaded", s3_key: s3Key, pipeline_triggered });
```

Upload is idempotent: if `transcript_s3_key` is already set, returns 200 without re-uploading. This makes it safe to retry.

Race condition handling: The transcript may arrive before or after `session.end` .

- If session is already `ended` when transcript arrives: pipeline triggers here
- If transcript arrives while session is still `detected` / `capturing` : pipeline triggers in the `session.end` handler when it detects `transcript_s3_key` is already set

4.3 Post-Processing Pipeline

The pipeline orchestrates everything after a session ends and its transcript is in S3.

```
// packages/core/src/session-pipeline.ts:93-323

export async function runSessionPipeline(
  deps: PipelineDeps,
  sessionId: string,
): Promise<PipelineResult> {

  // Step 1: Validate session state
  // Must be lifecycle="ended" with a transcript_s3_key
  if (session.lifecycle !== "ended") return error("not in ended state");
  if (!session.transcript_s3_key) return error("no transcript in S3");

  // Step 2: Claim session for parsing
  await sql`UPDATE sessions SET parse_status = 'parsing' WHERE id = ${sessionId}`;

  // Step 3: Download transcript from S3
  const transcriptContent = await s3.download(session.transcript_s3_key);

  // Step 4: Parse JSONL into structured data (see section 4.4)
  const parseResult = await parseTranscript(sessionId, transcriptContent);

  // Step 5: Persist to Postgres in a transaction (see section 4.5)
  await sql.begin(async (tx) => {
    // Clear any previously parsed data (idempotent re-run)
    await tx`DELETE FROM content_blocks WHERE session_id = ${sessionId}`;
    await tx`DELETE FROM transcript_messages WHERE session_id = ${sessionId}`;
    // Batch insert messages in chunks of 500
    await batchInsertMessages(tx, parseResult.messages);
    // Batch insert content blocks in chunks of 500
    await batchInsertContentBlocks(tx, parseResult.contentBlocks);
  });

  // Step 6: Advance lifecycle to 'parsed' with computed stats
  await transitionSession(sql, sessionId, "ended", "parsed", {
    parse_status: "completed",
    initial_prompt: extractInitialPrompt(parseResult.messages,
    parseResult.contentBlocks),
    duration_ms: stats.duration_ms,
    total_messages: stats.total_messages,
    user_messages: stats.user_messages,
    assistant_messages: stats.assistant_messages,
    tool_use_count: stats.tool_use_count,
    thinking_blocks: stats.thinking_blocks,
    subagent_count: stats.subagent_count,
    tokens_in: stats.tokens_in,
    tokens_out: stats.tokens_out,
    cache_read_tokens: stats.cache_read_tokens,
    cache_write_tokens: stats.cache_write_tokens,
```

```

    cost_estimate_usd: stats.cost_estimate_usd,
  });

  // Step 7: Generate LLM summary (best-effort – failure does NOT regress lifecycle)
  const summaryResult = await generateSummary(
    parseResult.messages, parseResult.contentBlocks, summaryConfig,
  );
  if (summaryResult.success && summaryResult.summary) {
    await transitionSession(sql, sessionId, "parsed", "summarized", {
      summary: summaryResult.summary,
    });
  }
  // If summary fails, session stays at 'parsed' – NOT 'failed'

  // Step 8: Upload parsed backup to S3 (best-effort, fire-and-forget)
  await s3.upload(buildParsedBackupKey(canonicalId, sessionId),
    JSON.stringify(parseResult), "application/json");
}

```

Never throws: all errors are captured in the returned `PipelineResult` .

4.4 JSONL Transcript Parser

The parser is a **pure function** — no I/O, no DB, no S3. Just data transformation.

```

// packages/core/src/transcript-parser.ts:73–269

export async function parseTranscript(
  sessionId: string,
  input: string | ReadableStream,
  options?: ParseOptions,
): Promise<ParseResult> {
  // Constants:
  // MAX_LINE_BYTES = 5MB per line
  // DEFAULT_MAX_INLINE_BYTES = 256KB for tool result content

  // Pass 1: Parse JSON, classify by type, group assistant lines by message.id
  // SKIP_TYPES: "progress", "file-history-snapshot", "queue-operation"
  // PROCESS_TYPES: "user", "assistant", "system", "summary"

  // Assistant message grouping:
  // Claude Code streams multi-line responses – multiple JSONL lines share
  // the same message.id. We group them and emit a single TranscriptMessage
  // per group, merging content blocks from all lines.

  // Pass 2: Build TranscriptMessage + ParsedContentBlock rows
  // Content block types: "text", "thinking", "tool_use", "tool_result"
  // Tool results exceeding 256KB are truncated with metadata.truncated = true

  // Pass 3: Compute stats
  // Token usage from the LAST line of each group (most complete in streaming)
  // Cost: $3.00/M input, $15.00/M output, $0.30/M cache read, $3.75/M cache write

```

```

    return { messages, contentBlocks, stats, errors, metadata };
}

```

Cost computation (packages/core/src/transcript-parser.ts:670-689):

```

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;

function computeCost(tokensIn, tokensOut, cacheRead, cacheWrite): number | null {
    return ((tokensIn ?? 0) * PRICE_INPUT_PER_MTOK +
            (tokensOut ?? 0) * PRICE_OUTPUT_PER_MTOK +
            (cacheRead ?? 0) * PRICE_CACHE_READ_PER_MTOK +
            (cacheWrite ?? 0) * PRICE_CACHE_WRITE_PER_MTOK) / 1_000_000;
}

```

Stats computed (TranscriptStats):

- total_messages , user_messages , assistant_messages
- tool_use_count , thinking_blocks , subagent_count (blocks where tool_name === "Task")
- tokens_in , tokens_out , cache_read_tokens , cache_write_tokens
- cost_estimate_usd , duration_ms (first to last timestamp)
- initial_prompt (first 1000 chars of first user message)

4.5 Persistence (Messages + Content Blocks)

Parsed data is persisted in two tables using batch INSERT with parameterized values.

```

// packages/core/src/session-pipeline.ts:333-448

// Batch size: 500 rows per INSERT (avoids exceeding Postgres parameter limits)
const BATCH_SIZE = 500;

// transcript_messages: 21 columns per row
// id, session_id, line_number, ordinal, message_type, role, model,
// tokens_in, tokens_out, cache_read, cache_write, cost_usd,
// compact_sequence, is_compacted, timestamp, raw_message (JSONB), metadata (JSONB),
// has_text, has_thinking, has_tool_use, has_tool_result

// content_blocks: 14 columns per row
// id, message_id, session_id, block_order, block_type,
// content_text, thinking_text, tool_name, tool_use_id, tool_input (JSONB),
// tool_result_id, is_error, result_text, metadata (JSONB)

// Uses sql.unsafe with numbered placeholders ($1, $2, ...) for batch efficiency

```

Idempotent re-parse: Before inserting, the transaction DELETES any existing content_blocks and transcript_messages for this session (CASCADE-safe).

4.6 LLM Summary Generation

After parsing, an LLM summary is generated (best-effort):

```
// packages/core/src/summary-generator.ts

// Calls Anthropic API (Claude Sonnet) with:
// - Parsed messages and content blocks
// - Configurable prompt template (summaryConfig)
// - Returns: { success: boolean, summary?: string, error?: string }

// On success: lifecycle transitions from "parsed" -> "summarized"
// On failure: session stays at "parsed" - NOT marked as failed
```

The `extractInitialPrompt` function captures the first 1000 characters of the first user message for quick display in session lists.

4.7 Pipeline Queue & Concurrency

To prevent unbounded memory growth, pipelines run through a bounded async work queue.

```
// packages/core/src/session-pipeline.ts:469-577

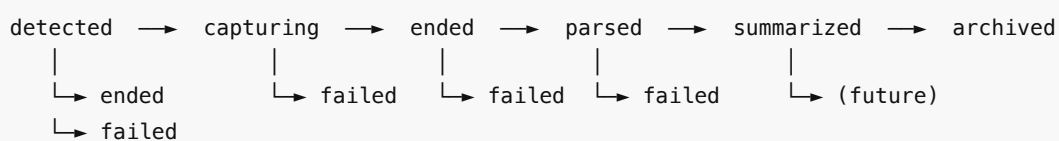
const MAX_QUEUE_DEPTH = 50; // Drop new entries beyond this depth

export function createPipelineQueue(maxConcurrent: number) {
  // maxConcurrent = 3 (set by server startup)
  // When a session is enqueued:
  //   - If queue depth < 50: add to pending list, try to process
  //   - If queue depth >= 50: log warning, drop the session
  // Processing:
  //   - While active < maxConcurrent && pending > 0: dequeue and run
  //   - Each pipeline run is fire-and-forget with .catch() error logging
  // stop():
  //   - Clears pending list
  //   - Waits for all in-flight pipelines to complete
}
```

Server startup (`packages/server/src/index.ts`) creates the queue with `maxConcurrent = 3` and `MAX_QUEUE_DEPTH = 50` .

5. Session Lifecycle

5.1 State Machine



failed → (terminal; use `resetSessionForReparse` to move back to ended)
archived → (terminal)

```
// packages/core/src/session-lifecycle.ts:58-66

export const TRANSITIONS: Record<SessionLifecycle, SessionLifecycle[]> = {
  detected: ["capturing", "ended", "failed"], // ended: short sessions skip
  capturing: ["ended", "failed"],
  ended: ["parsed", "failed"],
  parsed: ["summarized", "failed"],
  summarized: ["archived"],
  archived: [], // terminal
  failed: [], // terminal (reset via special
function)
};
```

5.2 Optimistic Locking Transitions

```
// packages/core/src/session-lifecycle.ts:127-219

export async function transitionSession(
  sql: Sql,
  sessionId: string,
  from: SessionLifecycle | SessionLifecycle[], // Can accept multiple source states
  to: SessionLifecycle,
  updates?: UpdatableSessionFields,
): Promise<TransitionResult> {
  // 1. Validate transition is legal (checked against TRANSITIONS map)
  // 2. Build dynamic UPDATE with SET clauses for lifecycle + extra columns
  // 3. Execute:
  //   UPDATE sessions SET lifecycle = $1, ... WHERE id = $2 AND lifecycle =
  ANY($3)
  //   RETURNING lifecycle
  //   ~~~~~ Optimistic lock: only one concurrent transition
  succeeds
  // 4. If no rows returned: diagnose (session not found vs wrong state)
}
```

Optimistic locking via `WHERE lifecycle = ANY($from)` ensures that concurrent transitions are serialized by Postgres — only one succeeds. This is critical for the pipeline, where multiple processes might try to transition the same session.

5.3 Recovery & Reparse

Stuck session detection (`packages/core/src/session-lifecycle.ts:419-448`):

```
export async function findStuckSessions(sql, stuckDurationMs = 600_000) {
  // Finds sessions where:
  //   lifecycle IN ('ended', 'parsed')
```

```
// parse_status IN ('pending', 'parsing')
// updated_at < now() - 10 minutes
// These are sessions whose parser crashed or timed out
}
```

Server startup (`packages/server/src/index.ts`) runs recovery 5 seconds after boot, re-triggering the pipeline for any stuck sessions.

Manual reparse (`resetSessionForReparse`):

```
// packages/core/src/session-lifecycle.ts:315-374

export async function resetSessionForReparse(sql, sessionId) {
  // In a transaction:
  // 1. DELETE content_blocks WHERE session_id = ?
  // 2. DELETE transcript_messages WHERE session_id = ?
  // 3. UPDATE sessions SET lifecycle='ended', parse_status='pending',
  //    summary=NULL, all_stats=NULL WHERE id=? AND lifecycle IN
  ('ended', 'parsed', 'summarized', 'failed')
  // Preserves the raw transcript_s3_key in S3
}
```

Exposed via `POST /api/sessions/:id/reparse` endpoint.

6. Real-Time Broadcasting

WebSocket Server

```
// packages/server/src/ws/index.ts

// Connection: ws://host/api/ws?token=<api_key>
// Auth: Token validated via constant-time comparison (same as HTTP Bearer)
// Ping/pong: 30s ping interval, 10s pong timeout → 40s max before eviction

// Subscription model:
// Client sends: { type: "subscribe", scope: "all" }
//               { type: "subscribe", workspace_id: "..." }
//               { type: "subscribe", session_id: "..." }

// Server broadcasts after consumer processes an event:
// { type: "event", event: Event }
// { type: "session.update", session: Session }
```

Broadcasting is fire-and-forget from the consumer — slow clients don't block event processing. The broadcaster filters messages to only reach clients with matching subscriptions.

Consumer Integration

```
// packages/server/src/pipeline/consumer.ts:169-184

// After successful event processing:
if (broadcaster) {
  broadcaster.broadcastEvent(entry.event);
  // Session lifecycle transitions:
  if (eventType === "session.start") {
    broadcaster.broadcastSessionUpdate(session_id, workspace_id, "detected");
  } else if (eventType === "session.end") {
    broadcaster.broadcastSessionUpdate(session_id, workspace_id, "ended");
  }
}
```

7. Database Schema Reference

Tables

workspaces (001_initial.sql)

Column	Type	Notes
id	TEXT PK	ULID
canonical_id	TEXT UNIQUE	Normalized git remote or local:<hash>
display_name	TEXT	Human-readable repo name
default_branch	TEXT	Set from first session.start hint
metadata	JSONB	Language, framework, all_remotes
first_seen_at	TIMESTAMPTZ	
updated_at	TIMESTAMPTZ	

devices (001_initial.sql)

Column	Type	Notes
id	TEXT PK	ULID
type	TEXT	CHECK: 'local' or 'remote'
name	TEXT	Hostname or friendly name
status	TEXT	CHECK: 'online','offline','provisioning','terminated'
hostname	TEXT	OS hostname
os	TEXT	OS name
arch	TEXT	CPU architecture
metadata	JSONB	CPU, memory details

first_seen_at	TIMESTAMPTZ	
last_seen_at	TIMESTAMPTZ	

workspace_devices (001_initial.sql + 004_...sql)

Column	Type	Notes
workspace_id	TEXT	Composite PK part 1
device_id	TEXT	Composite PK part 2
local_path	TEXT	Where workspace is checked out
hooks_installed	BOOLEAN	CC hooks installed
git_hooks_installed	BOOLEAN	Git hooks installed
pending_git_hooks_prompt	BOOLEAN	Should CLI prompt for git hooks?
git_hooks_prompted	BOOLEAN	Was user already prompted?
last_active_at	TIMESTAMPTZ	

sessions (001_initial.sql)

Column	Type	Notes
id	TEXT PK	CC session ID (from Claude Code)
workspace_id	TEXT FK	
device_id	TEXT FK	
lifecycle	TEXT	CHECK: detected/capturing/ended/parsed/summarized/archived/failed
parse_status	TEXT	CHECK: pending/parsing/completed/failed
started_at	TIMESTAMPTZ	
ended_at	TIMESTAMPTZ	
end_reason	TEXT	exit/clear/logout/crash
duration_ms	INTEGER	Computed from timestamps if hook sends 0
git_branch	TEXT	Branch at session start
model	TEXT	Claude model used
source	TEXT	startup/resume/clear/compact
transcript_s3_key	TEXT	S3 key for raw JSONL
parse_error	TEXT	Error message if parsing failed
summary	TEXT	LLM-generated summary

initial_prompt	TEXT	First 1000 chars of first user message
total_messages	INTEGER	Parsed stat
user_messages	INTEGER	Parsed stat
assistant_messages	INTEGER	Parsed stat
tool_use_count	INTEGER	Parsed stat
thinking_blocks	INTEGER	Parsed stat
subagent_count	INTEGER	Parsed stat (tool_name="Task")
tokens_in	INTEGER	Parsed stat
tokens_out	INTEGER	Parsed stat
cache_read_tokens	INTEGER	Parsed stat
cache_write_tokens	INTEGER	Parsed stat
cost_estimate_usd	NUMERIC	Parsed stat
tags	TEXT[]	User-applied tags
metadata	JSONB	

events (001_initial.sql)

Column	Type	Notes
id	TEXT PK	ULID
type	TEXT	One of 14 event types
timestamp	TIMESTAMPTZ	Client-side when event occurred
device_id	TEXT FK	
workspace_id	TEXT FK	Resolved ULID (not canonical string)
session_id	TEXT FK	NULL for git hooks outside sessions
data	JSONB	Type-specific payload
blob_refs	JSONB	S3 references array
ingested_at	TIMESTAMPTZ	Server-side timestamp

transcript_messages (002_transcript_tables.sql)

Column	Type	Notes
id	TEXT PK	ULID
session_id	TEXT FK (CASCADE)	

line_number	INTEGER	1-based JSONL line
ordinal	INTEGER	Conversation order
message_type	TEXT	user/assistant/system/summary
role	TEXT	
model	TEXT	
tokens_in	INTEGER	
tokens_out	INTEGER	
cache_read	INTEGER	
cache_write	INTEGER	
cost_usd	NUMERIC	Per-message cost
compact_sequence	INTEGER	Context compaction tracking
is_compacted	BOOLEAN	
timestamp	TIMESTAMPTZ	
raw_message	JSONB	Full raw line (lossless)
metadata	JSONB	
has_text	BOOLEAN	Denormalized flags
has_thinking	BOOLEAN	
has_tool_use	BOOLEAN	
has_tool_result	BOOLEAN	

content_blocks (002_transcript_tables.sql)

Column	Type	Notes
id	TEXT PK	ULID
message_id	TEXT FK (CASCADE)	
session_id	TEXT FK (CASCADE)	
block_order	INTEGER	Order within message
block_type	TEXT	CHECK: text/thinking/tool_use/tool_result
content_text	TEXT	For text blocks
thinking_text	TEXT	For thinking blocks
tool_name	TEXT	For tool_use blocks
tool_use_id	TEXT	Tool invocation ID

tool_input	JSONB	Tool arguments
tool_result_id	TEXT	Links result to tool_use
is_error	BOOLEAN	Tool result was error
result_text	TEXT	Tool result content (truncated at 256KB)
result_s3_key	TEXT	S3 key for large results (future)
metadata	JSONB	

git_activity (003_create_git_activity.sql)

Column	Type	Notes
id	TEXT PK	Same ULID as the event
workspace_id	TEXT FK	
device_id	TEXT FK	
session_id	TEXT FK	NULL for orphan git events
type	TEXT	CHECK: commit/push/checkout/merge
branch	TEXT	
commit_sha	TEXT	
message	TEXT	
files_changed	INTEGER	
insertions	INTEGER	
deletions	INTEGER	
timestamp	TIMESTAMPTZ	
data	JSONB	Author info, file list, etc.
created_at	TIMESTAMPTZ	

Key Indexes

```
-- Sessions
idx_sessions_workspace(workspace_id, started_at DESC)
idx_sessions_device(device_id, started_at DESC)
idx_sessions_lifecycle(lifecycle)
idx_sessions_tags USING GIN(tags) -- Array containment
idx_sessions_needs_recovery(lifecycle, updated_at) -- Partial: lifecycle IN
(ended,parsed)

-- Events
idx_events_workspace_time(workspace_id, timestamp DESC)
```

```
idx_events_session(session_id, timestamp ASC)           -- Partial: session_id IS
NOT NULL
idx_events_type(type, timestamp DESC)
idx_events_device(device_id, timestamp DESC)

-- Transcripts
idx_transcript_msg_session(session_id, ordinal)
idx_content_blocks_message(message_id, block_order)
idx_content_blocks_tool(tool_name)                     -- Partial: tool_name IS
NOT NULL
idx_content_blocks_text USING GIN(to_tsvector('english', content_text)) -- Full-
text search

-- Git Activity
idx_git_activity_workspace(workspace_id)
idx_git_activity_session(session_id)                   -- Partial: session_id IS
NOT NULL
idx_git_activity_timestamp(timestamp DESC)
idx_git_activity_workspace_time(workspace_id, timestamp DESC)
```

8. Error Handling & Resilience

CLI Layer — Zero Data Loss Guarantee

Failure Point	Handling	Exit Code
Config not found	Queue with hardcoded path	0
Invalid JSON data	Wrap as { _raw: ... }	0
Backend unreachable	Fall through to local queue	0
Queue write fails	Log to stderr, event lost	0
All paths	Never throw, never non-zero exit	0

Git Hook Safety

Condition	Action
fuel-code not in PATH	Exit 0, log to ~/.fuel-code/hook-errors.log
git_enabled: false in .fuel-code/config.yaml	Exit 0, no event
Workspace resolution fails	Exit 0, no event
fuel-code emit fails	Background process fails silently

Server Consumer Resilience

Failure Type	Action	Limit
Event processing error	Retry, increment counter	3 attempts

Redis NOGROUP (restart)	Re-create group, retry immediately	Infinite
Redis connection error	Wait 5s, retry	Infinite
Permanent event failure	Dead-letter (ack + error log)	After 3

Queue Drain Resilience

Failure Type	Action	Limit
Network timeout	Increment <code>_attempts</code> , stop batch	100 attempts
401 (invalid API key)	Stop drain immediately	100 attempts
503 (backend unavailable)	Stop drain	100 attempts
Corrupted queue file	Move to dead-letter	Immediate

Pipeline Resilience

Failure Point	Session State	Recovery
S3 download fails	failed	Manual reparse after S3 fix
Parse error (line-level)	Continues parsing	Partial results persisted
Postgres insert fails	failed	Manual reparse
Lifecycle transition race	Stays at current state	Other process won
Summary generation fails	Stays at parsed	NOT marked as failed
S3 backup upload fails	Stays at current	Ignored (best-effort)

Deduplication

ULID event IDs guarantee idempotency throughout the system:

- `INSERT ... ON CONFLICT (id) DO NOTHING` on `events` table
- `INSERT ... ON CONFLICT (id) DO NOTHING` on `git_activity` table
- Transcript upload returns `already_uploaded` if `transcript_s3_key` is set
- Queue drain reports duplicates from server response

9. Future Architecture

9.1 Phase 5: Remote Dev Environments

Status: Fully designed (`tasks/phase-5/dag.md`), 15 tasks, not yet started.

Phase 5 adds disposable cloud development environments on EC2. The architecture extends the existing abstractions without modifying them:

New database tables (planned):

- `remote_envs` — EC2 instance tracking (ID, status, IP, SSH key reference)

- `blueprints` — Saved environment configurations
- FK: `sessions.remote_env_id` → `remote_envs.id`

New event types (already defined in `EventType` union):

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

New CLI commands (planned):

- `fuel-code blueprint detect` — Auto-detect from `.fuel-code/env.yaml`
- `fuel-code remote up` — Provision EC2 instance
- `fuel-code remote ssh` — Connect to remote
- `fuel-code remote ls` — List remote environments
- `fuel-code remote down` — Terminate

Key design decisions:

- EC2 client as interface + mock boundary (testable without AWS)
- Provisioning is server-side (CLI polls progress via WebSocket)
- SSH keys: ephemeral, one-time download, cleaned up on termination
- Provisioning orchestrator as state machine (INIT → KEYS → SG → INSTANCE → TAGGED → DONE)
- Lifecycle enforcer: TTL check, idle timeout, provisioning timeout, orphan sweep
- Devices are symmetric: remote and local devices use the same pipeline

What already exists that Phase 5 builds on:

- Event pipeline (POST → Redis → processor → Postgres) — handles `remote.*` events
- WebSocket broadcasting — real-time provisioning progress
- S3 client — SSH key storage
- Handler registry — register `remote.*` handlers

9.2 V2: Analysis Layer

Status: Design only, no implementation.

The analysis layer reads from V1 tables (immutable) and writes to new `analysis_*` tables. V1 abstractions require **zero changes** to support V2.

V1 Tables (untouched):

`sessions, transcript_messages, content_blocks, events, git_activity`

|

▼ (read-only access)

V2 Analysis Engine (packages/analysis/ — future):

- Prompt extractor: identifies reusable prompts from transcripts
- 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 Claude Code skills from patterns

|

▼ (writes)

V2 Tables (new, additive):

`analysis_prompts, analysis_workflows, analysis_clusters, analysis_skills`

|

▼

New API routes: GET /api/analysis/prompts, /workflows, ...
New CLI/Web views: analysis dashboard, prompt library, skill generator

Why V1 already supports V2:

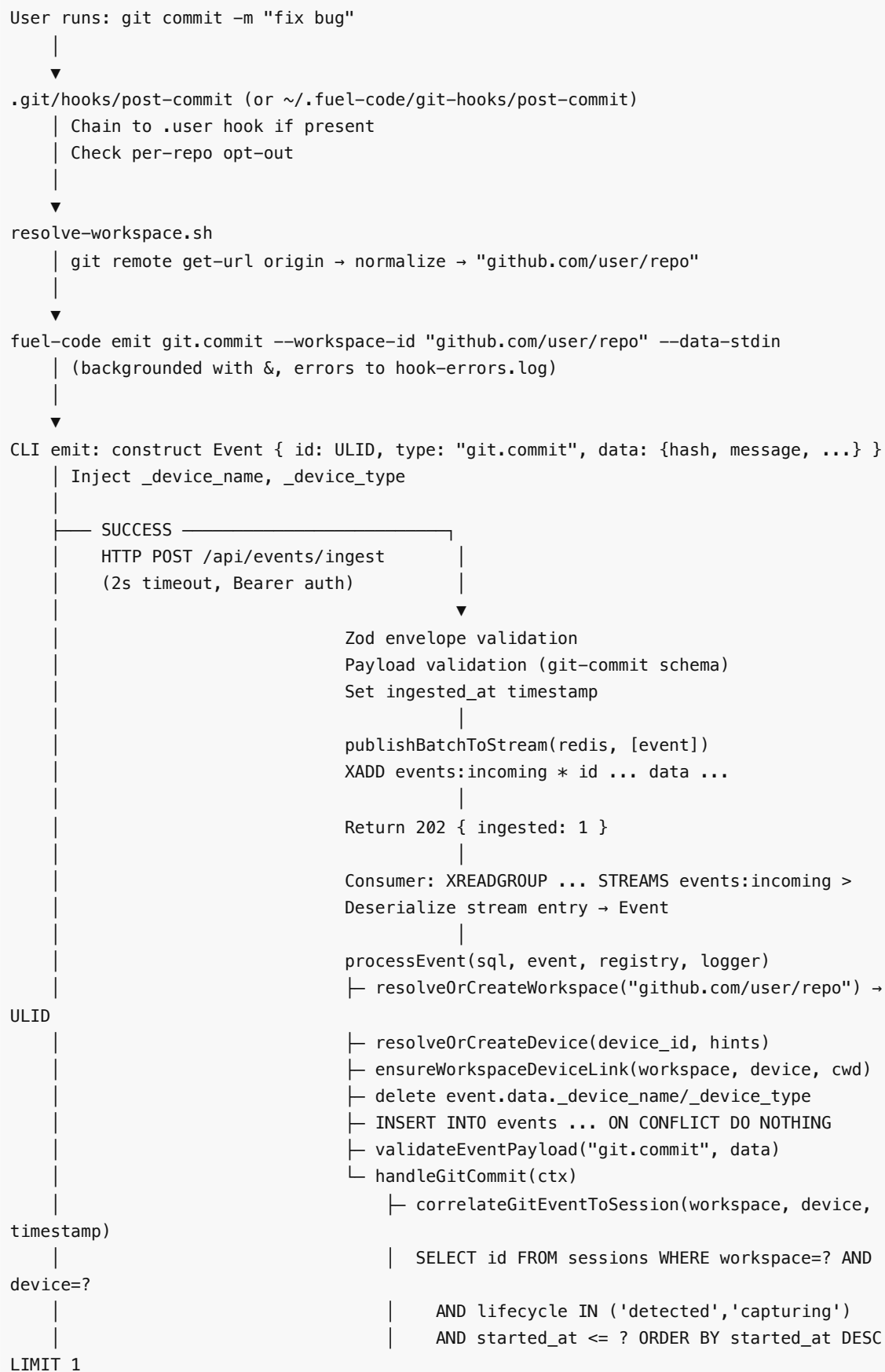
- transcript_messages + content_blocks provide rich structured data for analysis
- sessions.tags (TEXT array with GIN index) can be populated by analysis
- metadata JSONB fields on sessions/messages can store analysis annotations
- Full-text search index on content_blocks.content_text enables prompt search
- raw_message JSONB preserves original data for lossless analysis
- cost_estimate_usd enables cost-aware workflow optimization

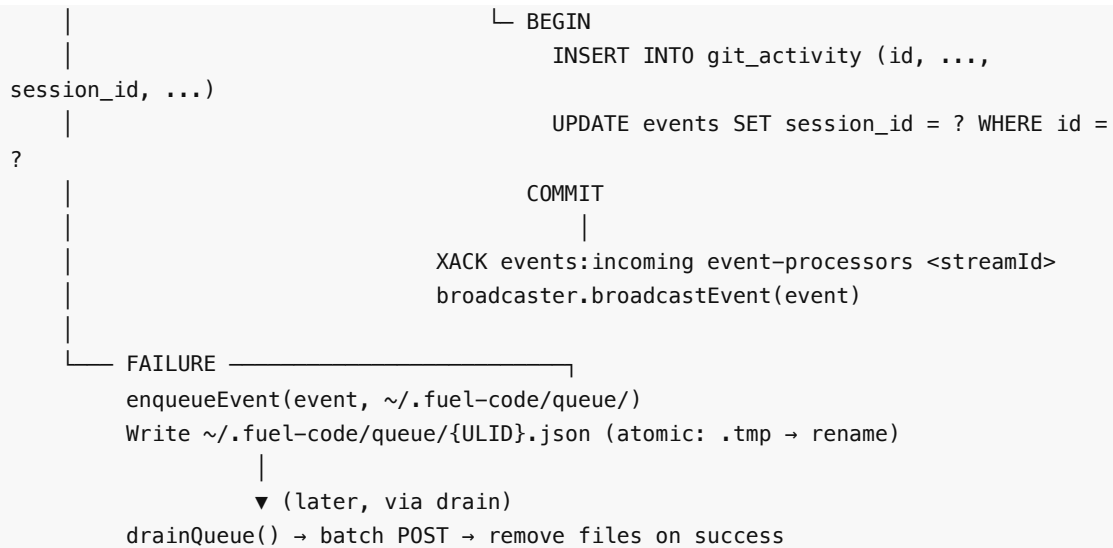
Appendix: Event Type Reference

Event Type	Source	Handler	Persisted To
session.start	CC SessionStart hook	handleSessionStart	sessions (INSERT)
session.end	CC SessionEnd hook	handleSessionEnd	sessions (UPDATE lifecycle)
session.compact	CC context compaction	(none registered)	events only
git.commit	post-commit hook	handleGitCommit	git_activity + events.session_id
git.push	pre-push hook	handleGitPush	git_activity + events.session_id
git.checkout	post-checkout hook	handleGitCheckout	git_activity + events.session_id
git.merge	post-merge hook	handleGitMerge	git_activity + events.session_id
remote.provision.start	(Phase 5)	(Phase 5)	events + remote_envs
remote.provision.ready	(Phase 5)	(Phase 5)	events + remote_envs
remote.provision.error	(Phase 5)	(Phase 5)	events + remote_envs
remote.terminate	(Phase 5)	(Phase 5)	events + remote_envs
system.device.register	fuel-code init	(none registered)	events only
system.hooks.installed	fuel-code hooks install	(none registered)	events only
system.heartbeat	(future)	(none registered)	events only

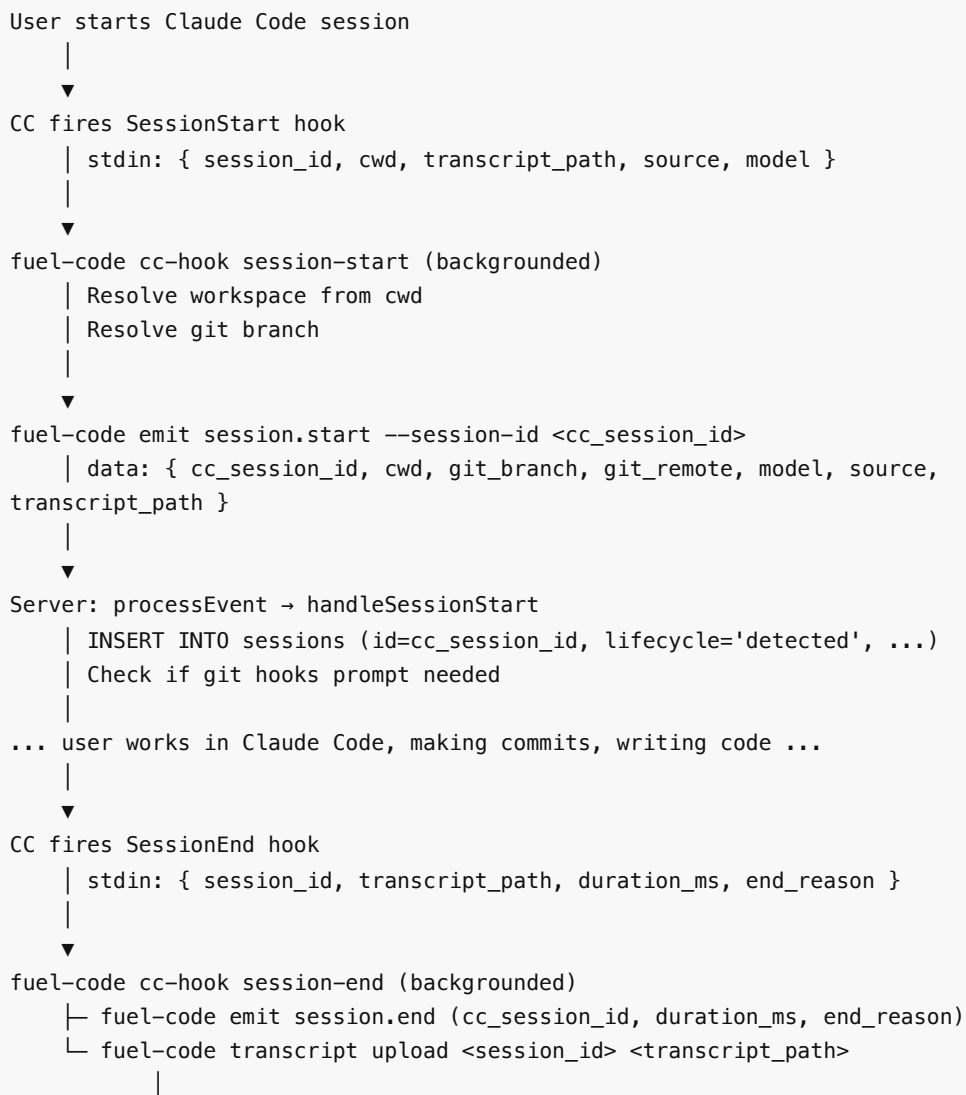
Appendix: Complete Data Flow Diagrams

Git Commit → Postgres





CC Session → Parsed Transcript



```

└─ emit session.end ───────────────────────────────────┐
    Server: processEvent → handleSessionEnd            │
    └─ Compute duration_ms if 0                        │
    └─ transitionSession(['detected','capturing'] → 'ended') │
        UPDATE sessions SET lifecycle='ended', ended_at=?, duration_ms=?
        WHERE id=? AND lifecycle = ANY(['detected','capturing'])
    └─ If transcript_s3_key already set:                │
        pipelineDeps.enqueueSession(sessionId)         │
└─ transcript upload ───────────────────────────────────┐
    POST /api/sessions/:id/transcript/upload           │
    └─ Validate Content-Length ≤ 200MB                 │
    └─ Check session exists                            │
    └─ Idempotency: already_uploaded? → 200           │
    └─ Build S3 key: transcripts/{canonical}/{session}/raw.jsonl
    └─ Buffer body → s3.upload()                       │
    └─ UPDATE sessions SET transcript_s3_key=?         │
    └─ If lifecycle='ended': trigger pipeline          │
                                                    ▼▼
runSessionPipeline(deps, sessionId)
└─ Verify lifecycle='ended' + has transcript_s3_key
└─ UPDATE sessions SET parse_status='parsing'
└─ s3.download(transcript_s3_key) → JSONL string
└─ parseTranscript(sessionId, content)
    └─ Split into lines, JSON.parse each
    └─ Skip: progress, file-history-snapshot, queue-
operation
    └─ Process: user, assistant, system, summary
    └─ Group assistant lines by message.id
    └─ Build TranscriptMessage[] + ParsedContentBlock[]
    └─ Compute stats (tokens, cost, duration, counts)
    └─ Return { messages, contentBlocks, stats, errors,
metadata }

|
└─ BEGIN TRANSACTION
    DELETE FROM content_blocks WHERE session_id=?
    DELETE FROM transcript_messages WHERE session_id=?
    Batch INSERT transcript_messages (500 rows/batch)
    Batch INSERT content_blocks (500 rows/batch)
    COMMIT
|
└─ transitionSession('ended' → 'parsed', { stats... })
|
└─ generateSummary(messages, blocks, config) [best-effort]
    └─ If success: transitionSession('parsed' →
'summarized')

|
└─ s3.upload(parsed-backup.json) [best-effort]
```