The Session Abstraction: Deep Dive

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

Table of Contents

1. What the Abstraction Provides
2. The Session Interface
3. The Lifecycle State Machine
4. The Database Schema
5. Type vs. Database Drift
6. Session Creation: The session.start Handler
7. Session Termination: The session.end Handler
8. The Post-Processing Pipeline
9. The Transcript Parser
10. The Summary Generator
11. The Pipeline Queue
12. Recovery: Stuck Sessions and Unsummarized Sessions
13. The Reset Mechanism
14. The Backfill System
15. The Transcript Upload Route
16. API Surface: REST Endpoints
17. CLI Surface
18. Hook Layer: How Sessions Enter the System
19. Cross-Abstraction Relationships
20. Gap Analysis: What's Missing or Misaligned
21. References

---

1. What the Abstraction Provides

The Session is the **primary unit of developer activity** in fuel-code. It represents one continuous Claude Code interaction — from the moment Claude Code launches to when it exits. Sessions are the organizing principle around which all other data is structured: events belong to sessions, transcripts belong to sessions, git activity is correlated to sessions, costs are computed per session, summaries describe sessions.

The Session is **one of the original five CORE.md abstractions** (Workspace, Session, Event, Device, Blueprint). CORE.md describes it as "a single Claude Code invocation — from start to end."

Core value proposition: Sessions provide the answer to the fundamental question: "What happened when I used Claude Code?" They aggregate raw event streams and parsed transcript data into a coherent, summarized, queryable unit with a lifecycle, statistics, and a human-readable description of what was accomplished.

What a Session captures:

- **Temporal bounds**: `started_at`, `ended_at`, `duration_ms`
- **Context**: which workspace, which device, which git branch, which model
- **Lifecycle state**: how far through processing this session has progressed
- **Transcript data**: parsed messages, content blocks, cost estimates

- **Derived intelligence**: LLM-generated summary, initial prompt, tool usage counts
- **User annotations**: tags for manual categorization

What a Session does NOT own (but correlates with):

- Git activity rows (linked via `session_id` FK in `git_activity`)
- Raw event rows (linked via `session_id` FK in `events`)
- The raw transcript blob (lives in S3, referenced by `transcript_s3_key`)

---

2. The Session Interface

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

```
export type SessionLifecycle =
  | "detected"
  | "capturing"
  | "ended"
  | "parsed"
  | "summarized"
  | "archived"
  | "failed";

export type ParseStatus = "pending" | "parsing" | "completed" | "failed";

export interface Session {
  id: string;                          // CC's session ID (NOT a ULID — comes from
Claude Code)
  workspace_id: string;                // FK to workspaces
  device_id: string;                   // FK to devices
  cc_session_id: string;               // Claude Code's own session ID (DRIFT — see
section 5)
  lifecycle: SessionLifecycle;         // Current state machine position
  parse_status: ParseStatus;           // Async parsing sub-state
  cwd: string;                         // Working directory
  git_branch: string | null;           // Branch at session start
  git_remote: string | null;           // Normalized remote URL (DRIFT — see section
5)
  model: string | null;                // Claude model used
  duration_ms: number | null;          // Total duration (null if still active)
  transcript_path: string | null;      // S3 key reference (DRIFT — see section 5)
  started_at: string;                  // ISO timestamp
  ended_at: string | null;             // ISO timestamp (null if still active)
  metadata: Record<string, unknown>;   // Extensible metadata
}
```

Key design decisions visible in this interface:

- **ID is from Claude Code**: The session's primary key IS the CC session ID. This enables natural deduplication — replaying events with the same `cc_session_id` is idempotent via `ON CONFLICT (id) DO NOTHING`.

- **Two independent state tracks**: `lifecycle` (session-level state machine) and `parse_status` (async parsing sub-state) are separate because parsing is a background job that doesn't block the session lifecycle from progressing.
- **17 fields in the type vs 29+ columns in the database**: The TypeScript type is intentionally a subset of what the database stores. The database has 11 statistical columns, tags, source, end_reason, parse_error, summary, initial_prompt, and more.

---

3. The Lifecycle State Machine

Defined in `packages/core/src/session-lifecycle.ts`. This is the formal state machine governing all session state transitions.

State Diagram

```
detected → capturing → ended → parsed → summarized → archived
    \                    \        \          \
     +→ ended (skip)      +→ failed +→ failed +→ failed
     +→ failed
```

Transition Map

```typescript
const TRANSITIONS: Record<SessionLifecycle, SessionLifecycle[]> = {
  detected:    ["capturing", "ended", "failed"],
  capturing:   ["ended", "failed"],
  ended:       ["parsed", "failed"],
  parsed:      ["summarized", "failed"],
  summarized:  ["archived"],
  archived:    [],     // terminal
  failed:      [],     // terminal — use resetSessionForReparse() to escape
};
```

What Each State Means

State	Meaning	Entered By	Can Transition To
`detected`	Session row created, CC is starting	`session.start` event handler	capturing, ended, failed
`capturing`	CC is actively running (SPEC ONLY — not used in practice yet)	Planned: `session.compact` event	ended, failed
`ended`	CC process exited, transcript ready for processing	`session.end` event handler	parsed, failed
`parsed`	Transcript parsed, stats computed, messages persisted	Session pipeline step 6	summarized, failed

`summarized`	LLM summary generated	Session pipeline step 7	archived
`archived`	Final state — parsed data can be pruned, recoverable from S3	Not yet implemented (planned)	(terminal)
`failed`	Error occurred at any stage	`failSession()` utility	(terminal — use `resetSessionForReparse()`)

The "detected → ended" Skip

Short sessions (e.g., user starts CC and immediately exits) can skip the `capturing` state entirely. The `session.end` handler accepts `["detected", "capturing"]` as valid source states, enabling both the normal path and the skip path.

Optimistic Locking

All transitions use `WHERE lifecycle = ANY($from)` in the SQL UPDATE. This means:

- Concurrent processes racing to transition the same session are serialized by Postgres — only one succeeds.
- If a transition fails, the function returns `{ success: false, reason: "Session is in state 'X', expected 'Y'" }` with diagnostic info.
- The `from` parameter can be a single state or an array of states, enabling flexible transition guards.

Updatable Fields

A lifecycle transition can atomically update additional columns alongside the state change:

```
type UpdatableSessionFields = Partial<{
  ended_at: string;
  end_reason: string;
  duration_ms: number;
  transcript_s3_key: string;
  parse_status: string;
  parse_error: string | null;
  summary: string;
  initial_prompt: string;
  total_messages: number;
  user_messages: number;
  assistant_messages: number;
  tool_use_count: number;
  thinking_blocks: number;
  subagent_count: number;
  tokens_in: number;
  tokens_out: number;
  cache_read_tokens: number;
  cache_write_tokens: number;
  cost_estimate_usd: number;
}>;
```

This design ensures that derived stats (from transcript parsing) and state transitions are atomic — no window where the session is in `parsed` state but stats haven't been written yet.

---

4. The Database Schema

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

```
CREATE TABLE sessions (
    id              TEXT PRIMARY KEY,         -- CC's session ID (text, not UUID)
    workspace_id    TEXT NOT NULL REFERENCES workspaces(id),
    device_id       TEXT NOT NULL REFERENCES devices(id),
    remote_env_id   TEXT,                     -- FK to remote_envs added in Phase 5
    lifecycle       TEXT NOT NULL DEFAULT 'detected'
                    CHECK (lifecycle IN
('detected','capturing','ended','parsed','summarized','archived','failed')),
    started_at      TIMESTAMPTZ NOT NULL,
    ended_at        TIMESTAMPTZ,
    end_reason      TEXT,
    initial_prompt  TEXT,
    git_branch      TEXT,
    model           TEXT,
    source          TEXT,                     -- "startup" | "resume" | "clear" |
"compact" | "backfill"
    transcript_s3_key TEXT,
    parse_status    TEXT DEFAULT 'pending'
                    CHECK (parse_status IN
('pending','parsing','completed','failed')),
    parse_error     TEXT,
    summary         TEXT,
    -- 11 statistical columns populated by transcript parsing:
    total_messages      INTEGER,
    user_messages       INTEGER,
    assistant_messages  INTEGER,
    tool_use_count      INTEGER,
    thinking_blocks     INTEGER,
    subagent_count      INTEGER,
    tokens_in           BIGINT,
    tokens_out          BIGINT,
    cache_read_tokens   BIGINT,
    cache_write_tokens  BIGINT,
    cost_estimate_usd   NUMERIC(10, 6),
    duration_ms         INTEGER,
    tags            TEXT[] NOT NULL DEFAULT '{}',
    metadata        JSONB NOT NULL DEFAULT '{}',
    created_at      TIMESTAMPTZ NOT NULL DEFAULT now(),
    updated_at      TIMESTAMPTZ NOT NULL DEFAULT now()
);
```

Indexes

```
-- List by workspace (most recent first)
CREATE INDEX idx_sessions_workspace ON sessions(workspace_id, started_at DESC);
-- List by device (most recent first)
CREATE INDEX idx_sessions_device ON sessions(device_id, started_at DESC);
-- Filter by lifecycle state
CREATE INDEX idx_sessions_lifecycle ON sessions(lifecycle);
-- Tag search using GIN (supports @> operator)
CREATE INDEX idx_sessions_tags ON sessions USING GIN(tags);
```

Notable Column Design Decisions

- **`id` is CC's session ID**: Not a ULID generated by fuel-code. This means session identity is determined by Claude Code, enabling natural deduplication.
- **`remote_env_id` placeholder**: Column exists with no FK constraint. The FK will be added in Phase 5 when the `remote_envs` table is created.
- **`source` column**: Tracks how the session was initiated. Values include `startup` (normal), `resume`, `clear`, `compact`, and `backfill` (historical ingestion).
- **`tags` as TEXT[]**: Postgres array with a GIN index, enabling `@>` containment queries. Tag mutation in the API supports three modes: replace, add, remove.
- **Token columns use BIGINT**: Anticipates sessions with billions of tokens (long-running sessions with many compactions).
- **`cost_estimate_usd` as NUMERIC(10,6)**: Precise to $0.000001, supporting sessions ranging from $0.003 to potentially $999.999999.

---

5. Type vs. Database Drift

The TypeScript `Session` interface (17 fields) does not match the database table (29+ columns). This is a known gap:

Issue	TypeScript Type	Database
`cc_session_id` exists in type	Present	**Not a separate column** — the CC session ID IS the `id` column
`cwd` exists in type	Present	**Not a column** — CWD is in the `session.start` event's `data.cwd`, not materialized
`git_remote` exists in type	Present	**Not a column** — the remote URL is on the workspace, not the session
`transcript_path` naming	Named `transcript_path`	Named `transcript_s3_key`
Missing from type	Not present	`end_reason`, `source`, `initial_prompt`, `summary`, `parse_error`, `tags`, all 11 stat columns, `remote_env_id`, `created_at`, `updated_at`

This drift means the `Session` interface is **not a faithful ORM-style mapping** of the sessions table. It's more of a "core identity" type. API responses return the full row (all 29+ columns), not the TypeScript interface shape.

---

6. Session Creation: The session.start Handler

File: `packages/core/src/handlers/session-start.ts`

When a `session.start` event is processed, the handler:

1. **Extracts fields** from `event.data`: `cc_session_id`, `git_branch`, `model`, `source`
2. **Inserts a session row** with `lifecycle = 'detected'`:

```
INSERT INTO sessions (id, workspace_id, device_id, lifecycle, started_at,
git_branch, model, source, metadata)
VALUES ($1, $2, $3, 'detected', $4, $5, $6, $7, '{}')
ON CONFLICT (id) DO NOTHING
```

3. **Checks for git hooks prompt** via `checkGitHooksPrompt()`:
   - If workspace is NOT `_unassociated` (i.e., is a git repo)
   - AND git hooks are NOT already installed for this workspace+device
   - AND user has NOT already been prompted
   - THEN flags `pending_git_hooks_prompt = true` on the `workspace_devices` row

Critical detail: The session ID used as the primary key is `cc_session_id` from the event data, NOT a fuel-code-generated ULID. This is what enables the `session.end` handler to find the session by ID.

Idempotency: `ON CONFLICT (id) DO NOTHING` means replaying the same `session.start` event is harmless.

---

7. Session Termination: The session.end Handler

File: `packages/core/src/handlers/session-end.ts`

When a `session.end` event is processed, the handler:

1. **Extracts fields**: `cc_session_id`, `end_reason`, `duration_ms`
2. **Computes duration** if missing: When `duration_ms` is 0 (hooks don't know real duration), computes it from `started_at` and the event timestamp.
3. **Transitions the session**: `["detected", "capturing"] → "ended"` via `transitionSession()` with optimistic locking. Atomically sets `ended_at`, `end_reason`, `duration_ms`.
4. **Triggers pipeline** (if available): If `ctx.pipelineDeps` is wired and the session already has a `transcript_s3_key` (backfill path), enqueues the session for post-processing.

The backfill path: In the normal live flow, the transcript upload happens AFTER `session.end`. But in the backfill flow, the transcript is uploaded before `session.end` is processed. The handler detects this (checks for `transcript_s3_key`) and triggers the pipeline immediately.

---

8. The Post-Processing Pipeline

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

The pipeline is the 8-step orchestrator that transforms a raw transcript blob into structured, queryable data:

```
Step 1: Validate session is in 'ended' state with transcript_s3_key
Step 2: Mark parse_status = 'parsing' (claim the session for processing)
Step 3: Download transcript JSONL from S3
Step 4: Parse JSONL into structured messages + content blocks
Step 5: Persist parsed data to Postgres (batched, 500 rows per INSERT)
Step 6: Advance lifecycle to 'parsed' with computed stats (atomic)
Step 7: Generate LLM summary (best-effort, non-blocking)
Step 8: Upload parsed backup to S3 (best-effort, fire-and-forget)
```

Key Design Decisions

- **Never throws**: All errors are captured in the `PipelineResult`. The pipeline always returns, never crashes.
- **Best-effort summary**: If summary generation fails (rate limit, timeout, API key issue), the session stays at `parsed`. This is NOT a failure — the session is still usable, just without a summary.
- **Idempotent persistence**: The transaction in step 5 starts with `DELETE FROM content_blocks/transcript_messages WHERE session_id = $1`, making re-runs safe.
- **Atomic stat writes**: Step 6 uses `transitionSession()` to atomically set lifecycle + all 11 stat columns + `initial_prompt` + `parse_status` in a single UPDATE.

The PipelineResult

```ts
interface PipelineResult {
  sessionId: string;
  parseSuccess: boolean;
  summarySuccess: boolean;
  errors: string[];
  stats?: TranscriptStats;
}
```

---

9. The Transcript Parser

File: `packages/core/src/transcript-parser.ts`

A **pure function** — no I/O, no DB, no S3. Takes raw JSONL text and returns structured data.

The 3-Pass Architecture

1. **Pass 1: Parse & Classify** — JSON-parse each line, classify by `type` field, group assistant lines by `message.id` (Claude Code streams multi-line responses)
2. **Pass 2: Build Messages & Content Blocks** — Emit `TranscriptMessage` + `ParsedContentBlock` rows in JSONL order. Assistant groups are merged into single messages.
3. **Pass 3: Compute Stats** — Aggregate token counts, cost estimates, tool usage, duration.

Line Type Handling

JSONL `type`	Action
`user`	One message per line. Content can be string (text) or array (tool_result).

`assistant`	Lines with same `message.id` are grouped → one message. Content blocks: text, thinking, tool_use, tool_result.
`system`	One message per line.
`summary`	One message per line (CC's own context compaction summaries).
`progress`, `file-history-snapshot`, `queue-operation`	Skipped entirely — CC bookkeeping.

Content Block Types

Block Type	What It Contains
`text`	Plain text output from the assistant
`thinking`	Extended thinking content
`tool_use`	Tool invocation: name, ID, input parameters
`tool_result`	Tool output: result text, is_error flag. Truncated to 256KB inline.

Cost Model

```
Input:  $3.00  / MTok
Output: $15.00 / MTok
Cache read:  $0.30  / MTok
Cache write: $3.75  / MTok
```

Per-message cost is computed and stored. Session-level cost is the sum of all message costs.

Size Limits

- Max JSONL line: 5 MB
- Max inline tool result: 256 KB (larger results are truncated with `metadata.truncated = true`)
- Initial prompt capture: first 1000 characters

---

10. The Summary Generator

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

Also a **pure function** (aside from the API call). Takes parsed messages + content blocks and returns a 1-3 sentence summary.

Prompt Engineering

The transcript is rendered into a condensed markdown format for prompting:

- **Header**: model, message count, tool use count, duration
- **User messages**: `[User]: {text}` (first 500 chars)
- **Assistant text**: `[Assistant]: {text}` (first 300 chars)
- **Tool uses**: `- Used {tool_name}` (name only, no input/output)
- **Thinking blocks**: skipped entirely

- **Tool results**: skipped entirely
- **Total limit**: 8000 chars (3000 head + 3000 tail with truncation marker)

System Prompt

```
You are a technical activity summarizer. Write a 1–3 sentence summary of this
Claude Code session in past tense. Focus on WHAT was accomplished, not HOW. Be
specific about files, features, or bugs. Do not start with "The user" or "This
session". Example: "Refactored the authentication middleware to use JWT tokens
and added comprehensive test coverage for the login flow."
```

Configuration

```typescript
interface SummaryConfig {
  enabled: boolean;
  model: string;           // Default: "claude-sonnet-4-5-20250929"
  temperature: number;     // Default: 0.3
  maxOutputTokens: number; // Default: 150
  apiKey: string;
}
```

Error Handling

The generator never throws. All failures are captured:

- Rate limit (429): returns `retryAfterSeconds` for the caller to use
- API timeout: 30 seconds via `AbortSignal.timeout()`
- Missing API key: returns error, doesn't attempt call
- Empty session: returns `"Empty session."` as the summary

---

11. The Pipeline Queue

The pipeline uses a **bounded async work queue** to limit concurrency:

```typescript
const queue = createPipelineQueue(maxConcurrent); // e.g., 3
queue.start(deps);
queue.enqueue("session-123"); // fire-and-forget
await queue.stop();           // waits for in-flight to finish
```

Constraints

- **Max concurrent pipelines**: Configurable (default: not hardcoded, set by server startup)
- **Max queue depth**: 50 pending sessions. Beyond this, new entries are dropped with a warning.
- **Graceful shutdown**: `stop()` clears the pending queue and returns a promise that resolves when all in-flight pipelines complete.

Why a Queue

Without a queue, a burst of `session.end` events (e.g., from backfill) would launch dozens of parallel S3 downloads, transcript parses, and LLM API calls. The queue prevents resource exhaustion and provides backpressure.

---

12. Recovery: Stuck Sessions and Unsummarized Sessions

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

Two independent recovery sweeps run on server startup (and optionally periodically):

Stuck Session Recovery (`recoverStuckSessions`)

Finds sessions stuck in intermediate pipeline states:

- **Stuck condition**: `lifecycle IN ('ended', 'parsed')` AND `parse_status IN ('pending', 'parsing')` AND `updated_at` older than threshold (default: 10 minutes)
- **With transcript**: Reset `parse_status` to `pending` , re-trigger pipeline via queue
- **Without transcript**: Transition to `failed` ("no transcript_s3_key available for reprocessing")
- **Limits**: Max 10 sessions per sweep (configurable)
- **Dry-run support**: Can report what would be recovered without making changes

Unsummarized Session Recovery (`recoverUnsummarizedSessions`)

Finds sessions stuck at `parsed` without a summary:

- **Stuck condition**: `lifecycle = 'parsed'` AND `parse_status = 'completed'` AND `summary IS NULL` AND `updated_at` older than threshold
- **Recovery**: Calls `resetSessionForReparse()` to move back to `ended` , then re-triggers the full pipeline (re-parse is fast, summary generation is what needs retrying)
- **Limits**: Max 10 sessions per sweep

---

13. The Reset Mechanism

`resetSessionForReparse()` in `packages/core/src/session-lifecycle.ts` provides a way to re-process a session from scratch:

What It Does (Atomically, in a Transaction)

1. `DELETE FROM content_blocks WHERE session_id = $1`
2. `DELETE FROM transcript_messages WHERE session_id = $1`
3. `UPDATE sessions SET lifecycle = 'ended', parse_status = 'pending', parse_error = NULL, summary = NULL, initial_prompt = NULL, [all 11 stat columns] = NULL WHERE id = $1 AND lifecycle IN ('ended', 'parsed', 'summarized', 'failed')`

What It Preserves

- The session row itself (ID, workspace, device, timestamps, tags, metadata)
- The `transcript_s3_key` (the raw blob in S3 is never deleted)
- Events in the `events` table (they reference the session but aren't owned by the parse)

Allowed Source States

- `ended` , `parsed` , `summarized` , `failed` — anything that has at least ended
- NOT allowed from `detected` or `capturing` — the session hasn't ended yet, there's nothing to re-parse

---

14. The Backfill System

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

The backfill system discovers and ingests ALL historical Claude Code sessions from `~/.claude/projects/`. This ensures no data loss — users get full visibility from their first CC session, not just from fuel-code installation.

Two Phases

Phase 1: Discovery (`scanForSessions`)

1. Scan `~/.claude/projects/` for project directories
2. Within each, find `{uuid}.jsonl` transcript files
3. Enrich metadata from `sessions-index.json` (when available) and JSONL headers
4. Resolve workspace via multi-strategy: git commands → parent directory walk → `projectDirToPath()` fallback → `_unassociated`
5. Skip: subagent directories, non-JSONL files, recently modified files (< 5 min, likely active sessions)

Phase 2: Ingestion (`ingestBackfillSessions`)

For each discovered session, via a concurrent worker pool (10 workers):

1. **Dedup check**: `GET /api/sessions/:id` — skip if already exists
2. **Emit synthetic `session.start`**: Creates the session row via the normal event pipeline
3. **Emit synthetic `session.end`**: Transitions session to `ended`
4. **Upload transcript**: `POST /api/sessions/:id/transcript/upload` — triggers the pipeline

Resilience Features

- **Rate limiting**: Shared backoff across all workers. When any worker hits 429, all workers pause.
- **Retry logic**: Transcript uploads retry up to 15 times with exponential backoff (handles 404 race conditions when the session.start event hasn't been processed yet).
- **Abort support**: `AbortSignal` propagation through all HTTP calls for clean Ctrl-C.
- **Resumable**: The `alreadyIngested` set skips previously completed sessions.
- **Progress reporting**: Callback with total/completed/skipped/failed counts.

The `projectDirToPath()` Resolver

Claude Code names project directories by replacing `/` with `-` in the absolute path. For example:

- `-Users-johnmemon-Desktop-fuel-code` → `/Users/johnmemon/Desktop/fuel-code`

The challenge: hyphens in original path components are indistinguishable from separators. The resolver uses a greedy approach that checks each prefix against the filesystem to find the longest valid path.

---

15. The Transcript Upload Route

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

`POST /api/sessions/:id/transcript/upload` — accepts raw JSONL body and streams to S3.

Behavior

1. **Validate Content-Length**: Must be present, max 200MB

2. **Validate session exists**: 404 if not found
3. **Idempotent**: If `transcript_s3_key` already set, returns `200 { status: "already_uploaded" }`
4. **Build S3 key**: `transcripts/{workspace_canonical_id}/{session_id}/raw.jsonl`
5. **Buffer and upload**: Buffers request body (bounded by 200MB limit), then uploads to S3
6. **Update session**: Sets `transcript_s3_key` on the session row
7. **Trigger pipeline**: If session lifecycle is `ended`, enqueues for post-processing
8. **Return**: `202 { status: "uploaded", s3_key, pipeline_triggered }`

Design Decision: Buffer Before Upload

The route buffers the request body before uploading to S3 (rather than streaming through). This decouples the client→server and server→S3 connections so a client disconnect mid-stream can't corrupt the S3 upload. Memory is bounded by the 200MB Content-Length check.

---

16. API Surface: REST Endpoints

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

List Sessions

`GET /api/sessions` with cursor-based pagination and filtering:

Parameter	Type	Description
`workspace_id`	string	Filter by workspace
`device_id`	string	Filter by device
`lifecycle`	string	Comma-separated lifecycle states
`after` / `before`	ISO date	Filter by started_at range
`ended_after` / `ended_before`	ISO date	Filter by ended_at range
`tag`	string	Filter by tag (uses GIN index)
`cursor`	string	Base64-encoded `{ s: started_at, i: id }` for keyset pagination
`limit`	number	Page size (default 50)

Response includes `sessions`, `next_cursor`, `has_more`.

Session Detail

`GET /api/sessions/:id` — Returns full session row JOINed with workspace name and device name.

Parsed Transcript

`GET /api/sessions/:id/transcript` — Returns `transcript_messages` with nested `content_blocks` aggregated as JSON arrays. Only available when `parse_status = 'completed'`.

Raw Transcript

`GET /api/sessions/:id/transcript/raw` — Returns presigned S3 URL (or redirects to it).

Session Events

`GET /api/sessions/:id/events` — All events with `session_id` matching, chronological order.

Session Git Activity

`GET /api/sessions/:id/git` — All `git_activity` rows correlated to this session, limit 500.

Update Session

`PATCH /api/sessions/:id` — Supports `summary` and three tag mutation modes:

- `tags` : Replace all tags
- `add_tags` : Append new tags (deduplicated)
- `remove_tags` : Remove specific tags

---

17. CLI Surface

The CLI exposes sessions through multiple commands (in `packages/cli/src/commands/`):

```
fuel-code sessions                      # List recent sessions
fuel-code sessions --workspace <name>   # Filter by workspace
fuel-code sessions --device <name>      # Filter by device
fuel-code sessions --today              # Today only
fuel-code session <id>                  # Detail view
fuel-code session <id> --transcript     # View parsed transcript
fuel-code session <id> --events         # View raw events
fuel-code session <id> --git            # Git activity during session
fuel-code session <id> --tag <tag>      # Add a tag
fuel-code session <id> --reparse        # Re-trigger transcript parsing
fuel-code backfill                      # Discover and ingest historical sessions
fuel-code backfill --status             # Show backfill progress
fuel-code backfill --dry-run            # Preview without ingesting
```

---

18. Hook Layer: How Sessions Enter the System

Sessions are born from Claude Code hooks installed at the user level (`~/.claude/settings.json`):

SessionStart Hook

Files: `packages/hooks/claude/SessionStart.sh` , `packages/hooks/claude/_helpers/session-start.ts`

Triggers on: CC process launch

1. Reads CC session ID from stdin/env
2. Resolves workspace canonical ID from `$CWD` (git remote lookup)
3. Reads device ID from `~/.fuel-code/device.json`

4. Calls `fuel-code emit session.start --data '{cc_session_id, cwd, git_branch, git_remote, model, source, transcript_path}'`
5. Must exit immediately (non-blocking — hooks can't delay CC startup)

SessionEnd Hook

Files: `packages/hooks/claude/SessionEnd.sh`, `packages/hooks/claude/_helpers/session-end.ts`

Triggers on: CC process exit

1. Reads CC session ID
2. Calls `fuel-code emit session.end --data '{cc_session_id, duration_ms, end_reason, transcript_path}'`
3. The CLI's emit command attempts an HTTP POST (2s timeout), falls back to local queue on failure
4. The hook also triggers transcript upload (separate HTTP call) — the critical data extraction step

---

19. Cross-Abstraction Relationships

```
Session
├── belongs_to → Workspace (workspace_id FK)
├── belongs_to → Device (device_id FK)
├── belongs_to → Remote Environment (remote_env_id FK, Phase 5)
├── has_many → Events (events.session_id FK, nullable)
├── has_many → Git Activity (git_activity.session_id FK, nullable)
├── has_many → Transcript Messages (transcript_messages.session_id FK)
│   └── has_many → Content Blocks (content_blocks.session_id + message_id FK)
├── references → S3 Blob (transcript_s3_key — raw JSONL)
├── references → S3 Blob (parsed backup — JSON)
├── generated_from → session.start Event
├── terminated_by → session.end Event
└── pipeline_triggered_by → transcript upload OR session.end (backfill path)
```

Session ↔ Event Correlation

Events have an optional `session_id` FK. When a `git.commit` event arrives:

1. The handler queries for an active session: `lifecycle = 'capturing'` in the same workspace + device
2. If found, sets `events.session_id` and `git_activity.session_id`
3. If not found, `session_id` stays NULL — the commit is workspace-level, not session-level

Session ↔ Workspace

Every session has exactly one workspace. The workspace is resolved during event processing from the git remote URL in the `session.start` event. The workspace provides display names and canonical IDs for the session.

Session ↔ Transcript

A session's transcript data lives in two places:

1. **S3**: The raw JSONL blob (`transcript_s3_key`). Source of truth. Never deleted.

2. **Postgres**: Parsed `transcript_messages` and `content_blocks`. Queryable. Can be pruned (archive) and recovered from S3.

---

20. Gap Analysis: What's Missing or Misaligned

1. TypeScript Type Drift (Medium Priority)

The `Session` interface has 17 fields while the database has 29+ columns. Fields like `cc_session_id`, `cwd`, `git_remote` exist in the type but not as separate DB columns. The type should be updated to match the DB or split into a `SessionRow` type.

2. `capturing` State Unused (Low Priority)

The `capturing` state exists in the state machine but is never entered in practice. The spec says `session.compact` events should transition to `capturing`, but the compact handler doesn't do this. Sessions go directly from `detected` to `ended`.

3. No `archived` Transition Logic (Low Priority)

The `summarized → archived` transition is defined but never triggered. There's no TTL-based archival, no pruning of parsed data, and no mechanism to restore archived sessions from S3 backup.

4. No Parse Dedup Claim (Medium Priority)

Step 2 of the pipeline sets `parse_status = 'parsing'` with a raw UPDATE (no WHERE clause on parse_status). This means two concurrent pipeline runs could both claim the same session. The optimistic locking on lifecycle catches this at step 6, but wasted work occurs. A `WHERE parse_status = 'pending'` would prevent this.

5. Summary Retry Is Expensive (Low Priority)

`recoverUnsummarizedSessions()` calls `resetSessionForReparse()` which deletes all parsed data and re-runs the full pipeline. This is safe but wasteful — the session just needs summary regeneration, not re-parsing. A targeted `retryOnlySummary()` function would be more efficient.

6. No Session Timeout for `detected` State (Medium Priority)

If a `session.start` event is processed but `session.end` never arrives (CC crashes without running the hook), the session stays at `detected` forever. There's no timeout mechanism to fail or clean up sessions that have been in `detected` state for longer than expected (e.g., 24 hours).

7. Backfill Race Condition Window (Low Priority)

During backfill, there's a window between emitting `session.start` and uploading the transcript where the session exists at `detected` without a transcript. If the server crashes during this window, the session becomes an orphan (no `session.end`, no transcript). The recovery sweep would eventually fail it, but a more robust approach would be to batch the entire session creation + upload atomically.

8. Tag Schema Has No Validation (Low Priority)

Tags are free-form strings with no validation on length, format, or reserved characters. The PATCH endpoint applies them directly to the TEXT[] column. A tag schema (max length, allowed characters, max tags per session) would prevent abuse.

9. No Concurrent Session Detection (Medium Priority)

If a user has two CC sessions running simultaneously in the same workspace on the same device, git commits during the overlap period are correlated to whichever session was most recently started. There's no heuristic to handle this case (e.g., checking if the commit's working directory matches one session's CWD more closely).

10. `parse_status` and `lifecycle` Can Desync (Low Priority)

There's no database constraint preventing `lifecycle = 'detected'` with `parse_status = 'completed'`, or `lifecycle = 'summarized'` with `parse_status = 'pending'`. The state machine logic prevents this in practice, but the DB schema doesn't enforce it.

---

21. References

File	What It Contains
`packages/shared/src/types/session.ts`	SessionLifecycle, ParseStatus, Session interface (17 fields)
`packages/core/src/session-lifecycle.ts`	TRANSITIONS map, transitionSession(), failSession(), resetSessionForReparse(), findStuckSessions(), getSessionState()
`packages/core/src/session-pipeline.ts`	runSessionPipeline() — 8-step post-processing orchestrator, createPipelineQueue()
`packages/core/src/session-recovery.ts`	recoverStuckSessions(), recoverUnsummarizedSessions()
`packages/core/src/session-backfill.ts`	scanForSessions(), ingestBackfillSessions(), projectDirToPath()
`packages/core/src/transcript-parser.ts`	parseTranscript() — 3-pass JSONL parser, cost model, content block conversion
`packages/core/src/summary-generator.ts`	generateSummary(), renderTranscriptForSummary(), extractInitialPrompt()
`packages/core/src/handlers/session-start.ts`	handleSessionStart(), checkGitHooksPrompt()
`packages/core/src/handlers/session-end.ts`	handleSessionEnd() — lifecycle transition + pipeline trigger
`packages/server/src/routes/sessions.ts`	7 REST endpoints: list, detail, transcript, raw, events, git, patch
`packages/server/src/routes/transcript-upload.ts`	POST upload — stream to S3, trigger pipeline
`packages/server/src/routes/timeline.ts`	Merged session + orphan git timeline
`packages/server/src/db/migrations/001_initial.sql`	Sessions table DDL, 4 indexes
`packages/cli/src/commands/sessions.ts`	CLI session list command

`packages/cli/src/commands/session-detail.ts`	CLI session detail command
`packages/cli/src/commands/backfill.ts`	CLI backfill command
`packages/hooks/claude/SessionStart.sh`	CC hook: session start
`packages/hooks/claude/SessionEnd.sh`	CC hook: session end
`packages/hooks/claude/_helpers/session-start.ts`	TS helper for session start event construction
`packages/hooks/claude/_helpers/session-end.ts`	TS helper for session end event construction
`tasks/CORE.md`	Session spec: lifecycle, payload schemas, pipeline, backfill