# The Git Activity Abstraction: Deep Dive

> *Last updated: 2026-02-27. Based on full codebase audit + analysis of git worktree edge cases.*

## Table of Contents

---

## 1. What the Abstraction Provides

Git Activity is a **denormalized materialized view** of git operations. Every git commit, push, checkout, and merge that happens in a repo where fuel-code hooks are installed gets captured, normalized, optionally correlated with an active Claude Code session, and stored in a dedicated Postgres table with 5 indexes optimized for the common query patterns.

Unlike the raw events table (which stores the original event payload as-is), `git_activity` flattens the type-specific fields (hash, message, branch, diff stats) into first-class columns. This means you can query "all commits on branch main this week" or "pushes during session X" without JSONB digging.

**Core value proposition**: Answer "what git work happened?" at three granularities:

- **Session-scoped**: "What did I commit/push during this CC session?"
- **Workspace-scoped**: "All git activity in this repo across all sessions"
- **Orphan**: "Git operations that happened outside any CC session"

The abstraction is **not one of the five CORE.md abstractions** (Workspace, Session, Event, Device, Blueprint). It's a derived, denormalized read-optimized projection of the raw `git.*` events, living in its own table. The originating event still exists in the `events` table; `git_activity` is the structured twin.

---

## 2. The 5-Stage Pipeline

Every git activity row passes through these stages:

```
Stage 1: Git Hook (bash)       — fires on git operation, extracts metadata, calls
`fuel-code emit`
Stage 2: CLI emit (bun)        — packages data into event envelope, POSTs to backend
(local queue fallback)
Stage 3: Backend Ingest        — HTTP route writes to events table, publishes to Redis
Stream
Stage 4: Event Processor       — Redis consumer dispatches to type-specific handler
Stage 5: Handler (core)        — correlates to session, INSERT into git_activity
(transactional)
```

### Stage 1 → 2: Hook → CLI

The hook scripts live in `packages/hooks/git/`. When `fuel-code hooks install` runs, it sets `git config --global core.hooksPath` to point at this directory. The hooks call `fuel-code emit <event-type>` with `--workspace-id` and `--data-stdin`, piping a JSON payload. The emit is backgrounded ( `&` ) so it never blocks git.

### Stage 2 → 3: CLI → Backend

`fuel-code emit` wraps the data in the standard event envelope ( `{id, type, device_id, workspace_id, timestamp, data}` ) and POSTs to `POST /api/events`. If the backend is unreachable, the event goes to a local SQLite queue ( `~/.fuel-code/queue.db` ) for later drain.

### Stage 3 → 4: Backend → Processor

The ingest route writes the event to the `events` table and publishes the event ID to a Redis Stream ( `events:stream` ). The consumer group reads from this stream and dispatches to the appropriate handler based on `event.type`.

### Stage 4 → 5: Processor → Handler

Each `git.*` event type has a dedicated handler that:

1. Calls `correlateGitEventToSession()` to find an active CC session
2. Wraps INSERT + event UPDATE in a Postgres transaction
3. Uses `ON CONFLICT (id) DO NOTHING` for idempotency

---

## 3. The Four Event Types

Defined in `packages/shared/src/types/git-activity.ts`:

```
export type GitActivityType = "commit" | "push" | "checkout" | "merge";
```

### Type: `commit`

**Hook**: `post-commit` (fires after `git commit` ) **What it captures**:

- `commit_sha` : full SHA from `git rev-parse HEAD`
- `message` : first 8192 chars of commit message
- `branch` : current branch name
- `files_changed` , `insertions` , `deletions` : from `git diff-tree --numstat`
- `data.author_name` , `data.author_email` : commit author
- `data.file_list` : array of `{path, status}` objects (A/M/D/R)

**Column mapping**: All first-class columns populated. Richest of the four types.

### Type: `push`

**Hook**: `pre-push` (fires before push executes — reads stdin, respects local hook exit codes) **What it captures**:

- `branch` : extracted from `refs/heads/<name>`
- `data.remote` : remote name (e.g., "origin")

- `data.commit_count` : number of commits being pushed
- `data.commits` : JSON array of pushed commit SHAs (max 100)

**Column mapping**: Only `branch` populated in first-class columns. `commit_sha` , `message` , `files_changed` , etc. are all NULL. Everything else is in `data` JSONB.

**Special behavior**: This is a **pre-** hook (not post-), so it fires before the push. If the local repo hook or user hook returns non-zero, the fuel-code hook exits with that code — it's the only hook that can (via delegation) block a git operation. But fuel-code's own logic always exits 0.

### Type: `checkout`

**Hook**: `post-checkout` (fires after branch switch) **What it captures**:

- `branch` : `to_branch ?? to_ref` (destination branch, or SHA if detached HEAD)
- `data.from_ref` , `data.to_ref` : source and target commit SHAs
- `data.from_branch` , `data.to_branch` : branch names (null for detached HEAD)

**Column mapping**: Only `branch` in first-class columns. No diff stats.

**Filter**: Only fires when `$3 == 1` (branch checkout), not file checkout.

### Type: `merge`

**Hook**: `post-merge` (fires after merge completes) **What it captures**:

- `commit_sha` : the merge commit hash
- `message` : merge commit message (first 4096 chars)
- `branch` : `into_branch` (the branch that received the merge)
- `files_changed` : diff stats between `HEAD^1` and `HEAD`
- `data.merged_branch` : the branch that was merged in (from `MERGE_HEAD` or commit message regex)
- `data.had_conflicts` : boolean (detected via `Conflicts:` in `MERGE_MSG` )

**Column mapping**: Uses `commit_sha` for the merge commit. `insertions` / `deletions` are NULL (only `files_changed` is populated).

---

## 4. The Bash Hook Layer

All four hooks follow the same structural template. Here's the invariant contract:

### Safety Invariants

```
1. ALWAYS exit 0 (except pre-push delegating to local hook)
2. Fire-and-forget: emit is backgrounded with `&`
3. Never block git — fuel-code failures are logged to ~/.fuel-code/hook-errors.log
4. Dispatch to local hooks first, chain to .user hooks
5. Per-repo opt-out via .fuel-code/config.yaml → git_enabled: false
```

### Hook Chaining

Each hook dispatches in this order:

```
# 1. Repo-local hook (core.hooksPath overrides .git/hooks, so we forward)
REPO_GIT_DIR=$(git rev-parse --git-dir 2>/dev/null)
LOCAL_HOOK="$REPO_GIT_DIR/hooks/post-commit"
# Only call if it exists, is executable, and ISN'T a fuel-code hook (prevent
recursion)
if [ -x "$LOCAL_HOOK" ] && ! head -5 "$LOCAL_HOOK" | grep -q "fuel-code:"
2>/dev/null; then
  "$LOCAL_HOOK" "$@" || true    # post-* hooks: swallow errors
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 (resolve workspace, extract data, emit)
```

### Double-Execution Prevention

The `grep -q "fuel-code:"` on line 5 of the local hook prevents recursion. If someone copies the fuel-code hook into `.git/hooks/`, it won't call itself.

### Data Extraction Patterns

**JSON escaping**: Hooks use `python3 -c 'import json,sys; print(json.dumps(sys.stdin.read()))'` for safe JSON encoding of commit messages and file paths. Falls back to raw string on python3 failure.

**Diff stats**: `git diff-tree --numstat -r HEAD` for commits, `git diff --numstat HEAD^1 HEAD` for merges.

**Stdin handling** (pre-push only): Must read ALL stdin before doing anything else — git expects it consumed:

```
PUSH_REFS=""
while IFS=' ' read -r local_ref local_sha remote_ref remote_sha; do
  PUSH_REFS+="${local_ref} ${local_sha} ${remote_ref} ${remote_sha}\n"
done
```

---

## 5. Workspace Resolution

`packages/hooks/git/resolve-workspace.sh` — called by every hook to determine which workspace this git operation belongs to.

### Normalization Rules

| Input Format | Output | Example |
|---|---|---|
| SSH remote | `host/user/repo` | `git@github.com:user/repo.git` → `github.com/user/repo` |

| | | |
|---|---|---|
| HTTPS remote | `host/user/repo` | `https://github.com/user/repo.git` → `github.com/user/repo` |
| No remote + commits | `local:<sha256>` | SHA-256 of the first commit hash |
| No remote + no commits | exit 1 (skip) | Empty repo, no tracking |

### Remote Priority

```
# Try origin first
REMOTE_URL=$(git remote get-url origin 2>/dev/null)
# Fallback: first remote alphabetically
if [ -z "$REMOTE_URL" ]; then
  FIRST_REMOTE=$(git remote 2>/dev/null | sort | head -1)
  REMOTE_URL=$(git remote get-url "$FIRST_REMOTE" 2>/dev/null)
fi
```

This mirrors `normalizeGitRemote()` in `packages/shared/src/canonical.ts` — the bash and TS implementations must produce identical canonical IDs for the same repo.

### Implications for Worktrees

Because `git remote get-url origin` returns the same value regardless of which worktree you're in (remotes are shared across linked worktrees), workspace resolution is **accidentally correct** for worktrees. Two linked worktrees of the same repo resolve to the same workspace ID. This is both a feature (activity groups together) and a gap (can't distinguish parallel work — see Section 9).

---

## 6. Session Correlation

`packages/core/src/git-correlator.ts` — the heuristic that links git operations to CC sessions.

### Algorithm

```
export async function correlateGitEventToSession(
  sql: Sql,
  workspaceId: string,
  deviceId: string,
  eventTimestamp: Date,
): Promise<CorrelationResult> {
  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 as string, confidence: "active" };
  }
  return { sessionId: null, confidence: "none" };
}
```

### Matching Criteria

1. **Same workspace + device**: The git event must come from the same repo on the same machine as the session
2. **Active lifecycle**: Session must be in `detected` or `capturing` state (not ended, not processed)
3. **Temporal ordering**: Session's `started_at` must be <= the event timestamp (no future sessions)
4. **Most recent wins**: `ORDER BY started_at DESC LIMIT 1` — if multiple sessions qualify, the newest one gets the correlation

### Confidence Levels

| Level | Meaning |
|---|---|
| `"active"` | Matched to a live session. The row gets `session_id` set. |
| `"none"` | No active session found. The row has `session_id = NULL` (orphan). |

### What's Missing

- **No `"ended"` confidence**: If a session just ended (lifecycle = `ended`) and a commit arrives 2 seconds later, it's orphaned even though the commit was almost certainly produced by that session.
- **No time-window fallback**: Unlike some activity trackers, there's no "within 5 minutes of an ended session" grace period.
- **Single-session assumption**: Only one session is matched per event. If two sessions are active on the same workspace+device (unlikely but possible), the newer one wins silently.

---

## 7. The Database Schema

`packages/server/src/db/migrations/003_create_git_activity.sql`:

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

### Indexes

```
CREATE INDEX idx_git_activity_workspace      ON git_activity(workspace_id);
CREATE INDEX idx_git_activity_session        ON git_activity(session_id) WHERE
session_id IS NOT NULL;
CREATE INDEX idx_git_activity_timestamp      ON git_activity(timestamp DESC);
CREATE INDEX idx_git_activity_type           ON git_activity(type);
CREATE INDEX idx_git_activity_workspace_time ON git_activity(workspace_id, timestamp
DESC);
```

Key design decisions:

- **Partial index on session_id**: Only indexes non-NULL values. Orphan lookups go through `workspace_id + timestamp` instead.
- **No composite index on (workspace_id, type)**: The `type` index is separate. Queries filtering by both rely on Postgres bitmap AND.
- **data JSONB is not indexed**: No GIN index. Type-specific fields that need querying are promoted to first-class columns.

### TypeScript Type

`packages/shared/src/types/git-activity.ts` :

```
export interface GitActivity {
  id: string;                          // ULID (same as originating event)
  workspace_id: string;
  device_id: string;
  session_id: string | null;           // null for orphan events
  type: GitActivityType;               // "commit" | "push" | "checkout" | "merge"
  branch: string | null;
  commit_sha: string | null;
  message: string | null;
  files_changed: number | null;
  insertions: number | null;
  deletions: number | null;
  timestamp: string;
  data: Record<string, unknown>;       // type-specific overflow
  created_at: string;
}
```

### Column Utilization by Type

| Column | commit | push | checkout | merge |
|---|---|---|---|---|
| `branch` | current branch | pushed branch | to_branch \|\| to_ref | into_branch |
| `commit_sha` | HEAD SHA | NULL | NULL | merge commit SHA |

| | | | | |
|---|---|---|---|---|
| `message` | commit message | NULL | NULL | merge message |
| `files_changed` | from numstat | NULL | NULL | from diff |
| `insertions` | from numstat | NULL | NULL | NULL |
| `deletions` | from numstat | NULL | NULL | NULL |
| `data` | author, file_list | remote, commits | from/to refs+branches | merged_branch, conflicts |

---

## 8. Query Surfaces (API + CLI + TUI)

### API Endpoints

**`GET /api/sessions/:id/git`** ( `packages/server/src/routes/sessions.ts:459` )

```
SELECT * FROM git_activity
WHERE session_id = $1
ORDER BY timestamp ASC
LIMIT 500
```

Returns `{ git_activity: [...] }`. Defensive LIMIT 500.

**`GET /api/workspaces/:id`** ( `packages/server/src/routes/workspaces.ts:260` ) Includes a git summary aggregate:

```
SELECT
  COUNT(*) FILTER (WHERE type = 'commit') AS total_commits,
  COUNT(*) FILTER (WHERE type = 'push') AS total_pushes,
  array_agg(DISTINCT branch) FILTER (WHERE branch IS NOT NULL) AS active_branches,
  MAX(timestamp) AS last_commit_at
FROM git_activity
WHERE workspace_id = $1
```

This gives workspace-level stats without needing to fetch all rows.

**`GET /api/timeline`** ( `packages/server/src/routes/timeline.ts:210+` ) The most complex surface. Merges three data sources:

1. **Sessions**: Each session item includes its correlated `git_activity[]` array
2. **Orphan git groups**: Git events with `session_id IS NULL` are grouped by workspace+device into `type: "git_activity"` timeline items
3. **Interleaving**: Both types are sorted by timestamp and paginated together

### CLI Commands

**`fuel-code session <id> --git`** ( `packages/cli/src/commands/session-detail.ts:482` ) Fetches `GET /api/sessions/:id/git` and renders a table:

```
HASH     MESSAGE            BRANCH   TIME          +/-       FILES
abc1234  feat: add auth     main     2m ago        +120 -3   5
def5678  fix: typo          main     5m ago        +1 -1     1
--- Other Activity ---
  push  main -> origin (2 commits)
  checkout  feature → main
```

Splits output into commits (table) and others (list below).

**`fuel-code session <id> --export json`** Includes `git_activity` array alongside session, transcript, and events in the export.

**`fuel-code session <id> --export md`** Generates a Markdown table of git activity.

**`fuel-code timeline`** ( `packages/cli/src/commands/timeline.ts` ) Renders session items with embedded git counts, orphan groups with their own section.

### TUI Components

**`GitActivityPanel`** ( `packages/cli/src/tui/components/GitActivityPanel.tsx` ) Two modes:

- **Sidebar mode** ( `detailed=false` ): Bullet list, max 10 items: `● abc1234 feat: add auth`
- **Full-width mode** ( `detailed=true` ): Per-commit stats: `● abc1234 feat: add auth [main]` with `(+120, -3, 5 files)` below

**`SessionDetailView`** ( `packages/cli/src/tui/SessionDetailView.tsx` ) Integrates GitActivityPanel in the sidebar when viewing a session.

---

## 9. Gap Analysis: Git Worktree Tracking

Git worktrees ( `git worktree add` ) create linked working trees that share the same repository (same `.git` directory, same remotes, same refs) but have independent working directories and HEADs. This creates several tracking blind spots.

### Gap 1: No Worktree Identifier in Payloads

**Severity: High**

None of the four hooks emit a worktree identifier. The event payload has `workspace_id` (from remote URL) and `device_id` (from machine), but no field distinguishing *which worktree* the operation happened in.

**Consequence**: If you have two linked worktrees open — one on `main` , one on `feature` — and commit in both simultaneously, both commits correlate to the same workspace+device. They're indistinguishable from sequential commits in a single worktree. There's no way to reconstruct "these 3 commits came from worktree A, those 2 from worktree B."

**Potential fix**: Each hook could emit `worktree_path` (from `git rev-parse --show-toplevel` ) and `is_linked_worktree` (detect `.git` file vs directory). The handler could store these in the `data` JSONB column.

### Gap 2: `--show-toplevel` Varies Per Worktree

**Severity: Medium**

Every hook uses `git rev-parse --show-toplevel` for the per-repo opt-out check:

```
REPO_ROOT=$(git rev-parse --show-toplevel 2>/dev/null)
if [ -f "$REPO_ROOT/.fuel-code/config.yaml" ]; then ...
```

In a linked worktree, `--show-toplevel` returns the *worktree's* path (e.g., `/Users/me/.claude/worktrees/abc`), NOT the main working tree path. A `.fuel-code/config.yaml` in the main repo root won't be found from a linked worktree — the opt-out silently fails.

**Potential fix**: Use `git rev-parse --path-format=absolute --git-common-dir` to find the shared `.git` directory, then resolve the main worktree root from there.

### Gap 3: `--git-dir` Returns Per-Worktree Private Dir

**Severity: Medium**

Hook chaining uses `git rev-parse --git-dir` to find local hooks:

```
REPO_GIT_DIR=$(git rev-parse --git-dir 2>/dev/null)
LOCAL_HOOK="$REPO_GIT_DIR/hooks/post-commit"
```

In the main worktree, `--git-dir` returns `.git` (the actual git directory). In a linked worktree, it returns `.git/worktrees/<name>` — a per-worktree private directory that typically does NOT contain a `hooks/` subdirectory. Hook chaining becomes a **no-op** in linked worktrees.

**Consequence**: If a repo has its own `.git/hooks/post-commit`, that hook will fire when committing in the main worktree but NOT when committing in a linked worktree. This is silent — no error, just missing local hook execution.

**Potential fix**: Use `git rev-parse --git-common-dir` to find the shared hooks directory:

```
COMMON_DIR=$(git rev-parse --git-common-dir 2>/dev/null)
LOCAL_HOOK="$COMMON_DIR/hooks/post-commit"
```

### Gap 4: `post-checkout` Fires on `git worktree add`

**Severity: Low-Medium**

`git worktree add ../my-worktree feature-branch` triggers the `post-checkout` hook with:

- `$1` = previous HEAD (from the originating worktree or 0000...0000)
- `$2` = new HEAD (the commit `feature-branch` points to)
- `$3` = 1 (branch flag)

Our hook records this as a normal branch checkout. There's no way to distinguish "user switched branches" from "user created a new worktree." The `from_ref` might be null/zeros, which is the only hint, but it's not checked or flagged.

**Consequence**: Phantom checkout events appear in the timeline. If Claude Code creates worktrees frequently (Claude Code's own `git worktree add` for isolation), these inflate the checkout count.

**Potential fix**: Detect worktree creation by checking if `$1` is all-zeros or if `GIT_DIR` points to a worktrees/ subdirectory. Add `data.is_worktree_creation: true` to differentiate.

### Gap 5: Session Correlation Ambiguity with Parallel Worktrees

**Severity: Medium-High**

The correlator matches on `workspace_id + device_id + lifecycle + timestamp`. If two CC sessions are running on the same repo (one per worktree — e.g., Claude Code in worktree A and Claude Code in worktree B), the correlator picks the **most recent** session by `started_at`.

This means commits from worktree A could be incorrectly attributed to the session running in worktree B (the newer session), purely based on start time ordering.

**Potential fix**: Add a `worktree_path` or `working_directory` field to both sessions and git events, then match on that during correlation. This would require changes to:

- Hook payloads (add `--cwd`)
- Session detection (add `working_directory` to sessions table)
- Correlator query (add `working_directory` match)

### Gap 6: `.git` File vs `.git` Directory

**Severity: Low**

In the main worktree, `.git` is a directory. In linked worktrees, `.git` is a **file** containing `gitdir: /path/to/main/.git/worktrees/<name>`. Any code that checks `if [ -d ".git" ]` (common in scripts) will fail in linked worktrees.

Our hooks don't directly check `.git` — they use `git rev-parse` commands — so this is mostly a theoretical concern. But session backfill code (`core/src/session-recovery.ts`) or any future path resolution that walks up directories looking for `.git` directories would need to handle the file case.

### Gap 7: Session Backfill Path Resolution

**Severity: Medium**

Claude Code sessions store a `cwd` (project directory) in their transcript. Session backfill uses this to find the workspace. If a CC session was started inside a linked worktree (e.g., `/Users/me/.claude/worktrees/abc-def`), and that worktree has since been cleaned up, the backfill path is unresolvable — the directory no longer exists, and there's no way to map it back to the original repo.

**Current behavior**: Backfill silently fails workspace detection for cleaned-up worktree paths.

**Potential fix**: Store both `cwd` and the resolved `workspace_id` (canonical remote) at session detection time, so backfill doesn't need to re-derive it from the filesystem.

### Summary Table

| Gap | Severity | Impact | Fix Complexity |
|---|---|---|---|
| No worktree ID in payloads | High | Can't distinguish parallel worktree activity | Low (add field) |
| `--show-toplevel` varies | Medium | Opt-out fails in linked worktrees | Low (use `--git-common-dir`) |
| `--git-dir` per-worktree | Medium | Hook chaining no-op in linked worktrees | Low (use `--git-common-dir`) |

| | | | |
|---|---|---|---|
| `post-checkout` on worktree add | Low-Med | Phantom checkout events | Low (detect zeros/$GIT_DIR) |
| Correlation ambiguity | Med-High | Wrong session attribution | Medium (add cwd matching) |
| `.git` file vs directory | Low | Future path resolution bugs | Low (handle both) |
| Backfill path resolution | Medium | Lost workspace for cleaned-up worktrees | Medium (store canonical ID early) |

---

## 10. References

| # | File | Description |
|---|---|---|
| 1 | `packages/shared/src/types/git-activity.ts` | TypeScript type definition |
| 2 | `packages/server/src/db/migrations/003_create_git_activity.sql` | Table schema + indexes |
| 3 | `packages/core/src/git-correlator.ts` | Session correlation heuristic |
| 4 | `packages/core/src/handlers/git-commit.ts` | Commit event handler |
| 5 | `packages/core/src/handlers/git-push.ts` | Push event handler |
| 6 | `packages/core/src/handlers/git-checkout.ts` | Checkout event handler |
| 7 | `packages/core/src/handlers/git-merge.ts` | Merge event handler |
| 8 | `packages/hooks/git/post-commit` | Bash hook: commit metadata extraction |
| 9 | `packages/hooks/git/pre-push` | Bash hook: push ref parsing |
| 10 | `packages/hooks/git/post-checkout` | Bash hook: branch switch detection |
| 11 | `packages/hooks/git/post-merge` | Bash hook: merge metadata extraction |
| 12 | `packages/hooks/git/resolve-workspace.sh` | Workspace canonical ID resolution |
| 13 | `packages/server/src/routes/sessions.ts:459` | API: session git activity endpoint |
| 14 | `packages/server/src/routes/workspaces.ts:260` | API: workspace git summary aggregate |
| 15 | `packages/server/src/routes/timeline.ts:210` | API: timeline merge (sessions + orphan git) |
| 16 | `packages/cli/src/commands/session-detail.ts:482` | CLI: `--git` flag handler |

| | | |
|---|---|---|
| 17 | `packages/cli/src/tui/components/GitActivityPanel.tsx` | TUI: git activity sidebar/full panel |
| 18 | `packages/cli/src/tui/SessionDetailView.tsx` | TUI: session detail with git sidebar |
| 19 | `packages/shared/src/canonical.ts` | TS workspace normalization (mirrors resolve-workspace.sh) |
| 20 | Git Worktree Docs | Official git worktree documentation |