会话持久化事件目录

English | 中文

[SessionEvent [][SessionEventMap [@deepseek-ai/dsh-session [@deepseek-ai/dsh-session/types 的声明合并，并附有源 JSDo
声明、surface se 标记和声明位置。本文档Jrface deriveMessages() 投影）、istence.md[如何让5志持久化）和d 中量成区J（5][session/event emit [

scripts/gen-persistence-catalog.ts [pnpm run verify-persistence-catalog doc-sync （文档同步门_ ts persistence-catalog 评审_
围栏（doc-typecheck 会跳过这些围[，3T 声ST 用了其所属模块中[类型）。payload 中的类型名称会链接到记录该类型的页面。参见

[type][seq][事件的pocl time data [ignorable][surfaceOp sourceEventSeqs °urface [SurfaceEventType 成员：它会生成一条
surface [o °-only 表示其他所有事件：这类记录可持久 Session.append][SESSION_FORMAT_VERSION = 0 ；这是两处布格式，不暗示任何兼容性（参见版本立场），范围仅限本仓库中的包；下游插件可以规约_

事件信封

```
/** The appendable event-type keys of {@link SessionEventMap}, plugin-merged extensions included. */
export type SessionEventType = keyof SessionEventMap

/**
 * The subset of {@link SessionEventType} values whose events produce LLM
 * messages and are eligible to appear on the ordered surface. Only these
 * event types may carry {@link SurfaceOp} and {@link SessionEvent.sourceEventSeqs}.
 */
export type SurfaceEventType =
  | 'user/message'
  | 'assistant/message'
  | 'tool/result'

/**
 * How a session event entered the ordered surface. Only valid on
 * {@link SurfaceEventType} events.
 *
 * - `'append'`: added to the tail — normal path for user/assistant/tool
 *   messages.
 * - `{ op: 'replace', start, end }`: replaces surface nodes from `start`
```

```
 *   (inclusive) through `end` (inclusive) with this node. Both must exist as
 *   surface nodes in the current surface. `start === end` replaces a single
 *   node. The node's {@link SessionEvent.sourceEventSeqs} must include every
 *   shadowed surface node. Used by compaction; any surface-replacing producer
 *   may use it.
 */
export type SurfaceOp =
  | 'append'
  | { op: 'replace'; start: SessionSeq; end: SessionSeq }

/**
 * One immutable entry in the session log.
 *
 * A proper discriminated union over `type` (not independent `type`/`data`
 * unions), so `switch (event.type)` narrows `event.data` without casts.
 *
 * The {@link sourceEventSeqs} and {@link surfaceOp} fields are conditional:
 * they only exist on {@link SurfaceEventType} variants (`user/message`,
 * `assistant/message`, `tool/result`).
 * Non-surface events (boundary markers, chunks, usage, errors) never carry
 * surface metadata — the compiler enforces this at `Session.append()`
 * call sites.
 */
export type SessionEvent<T extends SessionEventType = SessionEventType> = {
  [K in SessionEventType]: {
    type: K
    /** Monotonic sequence number within the session. */
    seq: SessionSeq
    /** Unix epoch milliseconds. */
    time: number
    data: SessionEventMap[K]
    /**
     * Marks an event a reader may safely skip when it does not recognize
     * `type`. Absent means required: a reader meeting an unrecognized type
     * without this marker MUST refuse to reconstruct the session instead of
     * silently dropping the event, because an unrecognized required event may
     * change how the rest of the log is interpreted. A writer sets `true` only
     * on purely informational records whose loss cannot affect reconstruction;
     * defaulting to required means a forgotten marker over-refuses (an
     * inconvenience) rather than silently resuming a gutted session.
```

```
     */
    ignorable?: true
  } & (K extends SurfaceEventType ? {
    /**
     * Seq numbers of earlier events that this event cites as sources
     * (e.g. the `assistant/chunk` seqs that built an `assistant/message`,
     * or the surface nodes shadowed by a compaction replace node). An
     * `assistant/message` may carry a present empty array for a known empty
     * provider stream; when the field is absent, the event does not record which
     * earlier events produced the message.
     */
    sourceEventSeqs?: SessionSeq[]
    /** How this event entered the surface; absent for non-surface events. */
    surfaceOp?: SurfaceOp
  } : object)
}[T]
```

packages/core/session/src/types.ts:366 · packages/core/session/src/types.ts:373 · packages/core/session/src/types.ts:402 · packages/core/session/src/types.ts:434

agent/*

agent/inbox/spliced — log-only

```
/**
 * One normalized mutation of an agent's durable pending-message lists.
 * Live dispatch precedes projection mutation, so synchronous observers may
 * read the pre-splice inbox to recover the removed messages.
 */
'agent/inbox/spliced': {
  target: InboxTarget
  start: number
  removedCount?: number
  inserted: UserMessage[]
  outcome?: 'canceled'
```

```
}
```

packages/core/agent/src/types.ts:58

agent-preset/*

agent-preset/selected — log-only

```
/**
 * The session's agent preset was chosen after creation, while the session
 * was still blank. Log-only: it records the composition later turns ran
 * under, so a resumed or forked session rebuilds the same one instead of
 * the header's creation-time value.
 */
'agent-preset/selected': { agentPreset: string }
```

packages/preset/agent-presets/src/session.ts:28

approval/*

approval/asked — log-only

```
/**
 * An approval question was put to the answerer chain — log-only audit
 * (like `hook/*`; NOT a surface event, carries no `surfaceOp`). `id` pairs
 * it with the `approval/decided` that always follows; `toolName` is the
 * tool the question is about, `callId` the exact tool call when the asker
 * had one, `reason` the asker's human-readable explanation (e.g. a hook's
 * permission-decision reason).
 */
'approval/asked': {
  id: ApprovalRequestId
  toolName: string
  callId?: ToolCallId
  reason?: string
}
```

ToolCallId

packages/interaction/user-approval/src/types.ts:44

`approval/decided` — log-only

```
/**
 * The outcome of a prior `approval/asked` (same `id`) — log-only audit.
 * Exactly one per ask, appended when the outcome is known: a decision, a
 * cancellation, or the fail-closed `'unavailable'`.
 */
'approval/decided': {
  id: ApprovalRequestId
  outcome: ApprovalOutcome
}
```

packages/interaction/user-approval/src/types.ts:55

`approval/policy` — log-only

```
/**
 * The session's approval policy was switched — log-only, durable,
 * replayable, never in the model transcript (the model learns the policy
 * from the runtime-context snapshot and live switch notices). The LAST
 * such event is the session's override.
 * `source: 'delegation'` marks an override seeded into a child; an absent
 * source is a runtime switch.
 */
'approval/policy': {
  policy: ApprovalPolicy
  /** Marks an override seeded into a child at delegation. */
  source?: 'delegation'
}
```

packages/interaction/user-approval/src/index.ts:32

`assistant/*`

`assistant/chunk` — log-only

```
/** Raw stream chunk — token-level replay fidelity. */
'assistant/chunk': { turn: number; step: number; chunk: StreamChunk }
```

StreamChunk

packages/core/session/src/types.ts:289

assistant/message — surface

```
/**
 * Assembled assistant message for one step (derived history uses this).
 * Carries the step's `usage` when the adapter reported token accounting, so
 * the model output and its accounting travel together (there is no separate
 * usage record). `usage` is absent when the adapter reported none. A turn
 * cancelled mid-stream finalizes its delivered text/reasoning prefix as this
 * event with `interrupted: true`; undispatched tool calls are absent. The
 * marker distinguishes that prefix without re-deriving interruption from turn
 * boundaries. An aborted turn with no such event streamed no visible content.
 */
'assistant/message': { turn: number; step: number; message: AssistantMessage; usage?: TokenUsage;
interrupted?: true }
```

TokenUsage

packages/core/session/src/types.ts:300

command/*

command/done — log-only

```
/**
 * The paired command settled. `kind`/`text` carry the handler's verbatim
 * outcome (a thrown/aborted handler settles as `kind: 'error'` with the
 * rendered failure). A successful command may identify the earlier
 * authoritative domain event for a richer client-computed presentation.
 */
'command/done': {
  commandId: CommandId
```

```
  kind: 'success' | 'error'
  text?: string
  sourceEventSeq?: import('@deepseek-ai/dsh-session/types').SessionSeq
}
```

packages/interaction/commands/src/types.ts:104

`command/run` — log-only

```
/**
 * A resolved slash command entered its handler. Log-only (never model
 * surface); paired with `command/done` by `commandId`, mirroring the
 * `tool/call`↔`tool/result` pairing. The payload is structured — `name`
 * and `args` are `parseCommand`'s own split (name and verbatim rawInput,
 * separator whitespace included), so a consumer (a projection unit
 * folding its own command records, a rich command card) never re-parses
 * a line. `args` is absent when the definition sets `recordInput: false`
 * because an authoritative domain event owns the input payload.
 */
'command/run': { commandId: CommandId; name: string; args?: string; source: CommandSource }
```

packages/interaction/commands/src/types.ts:97

`compaction/*`

`compaction/end` — log-only

```
/**
 * Marks the end of a compaction — log-only, releases the lock. Its owner
 * matches `compaction/start`; `error` records an unsuccessful attempt.
 */
'compaction/end': { compactionId: CompactionId; sourceCommandId?: CommandId; turn: number | null; error?:
string }
```

packages/compaction/compaction/src/types.ts:72

`compaction/prune` — log-only

```
/**
 * Shadow price of one model-free prune replacement — log-only, no
 * surfaceOp. The shared shadow-price protocol: a surface `replace` event
 * is priced by the metering event immediately before it (`compaction/summary`
 * for a summarizing compaction, this event for a prune), which states the
 * heuristic token price of the exact replaced range so a pure consumer
 * can subtract it without retaining per-node prices. The replacement MUST
 * be appended synchronously right after this event.
 */
'compaction/prune': {
  /** The replaced range's first and last surface-node seqs (a surface-position span, like {@link
CompactionResult.shadowedRange}). */
  shadowedRange: { start: SessionSeq; end: SessionSeq }
  /** The seqs of all shadowed surface nodes, in surface order. */
  shadowedSeqs: SessionSeq[]
  /** Heuristic price of the shadowed content under the token-meter's fixed estimator. */
  shadowedTokenCount: number
}
```

packages/compaction/compaction/src/types.ts:82

`compaction/start` — log-only

```
/**
 * Marks the start of a compaction — log-only, holds the lock until
 * `compaction/end`. A numbered owner is strictly enclosed by that open turn;
 * `null` identifies a standalone manual transaction between turns.
 */
'compaction/start': { compactionId: CompactionId; sourceCommandId?: CommandId; turn: number | null }
```

packages/compaction/compaction/src/types.ts:24

`compaction/summary` — log-only

```
/**
 * Completed summary, its inputs, and its model call facts — log-only, no surfaceOp.
 * The summary content is in `data.summary`; the actual surface replacement
 * is performed by the immediately following `user/message` event that
 * shadows the compacted range. That adjacency is contractual — the
```

```
 * shadowed pricing fields are the replacement's shadow price, so a
 * consumer may pair a replacement with the metering event directly
 * before it (`compaction/prune` documents the shared protocol).
 */
'compaction/summary': {
  compactionId: CompactionId
  sourceCommandId?: CommandId
  summary: ContentBlock[]
  shadowedRange: { start: SessionSeq; end: SessionSeq }
  shadowedSeqs: SessionSeq[]
  shadowedTokenCount: number
  /** The provider route that wrote the summary. */
  provider: string
  /**
   * The model that wrote the summary — the summarize call's envelope,
   * reported by the backend that made the call, logged so the one-shot
   * request is reconstructable from log + code and "which model wrote
   * this summary" has a durable answer (the reconstructability Agent Note).
   */
  model: string
  /** The generation cap the summarize call sent, when one applied. */
  maxTokens?: number
  /** Provider-reported token usage for the summarization request, when emitted. */
  usage?: TokenUsage
} & (
  | {
    /** Complete provider output before the backend's safe summary projection. */
    rawOutput: ContentBlock[]
    /** Identifies exactly one call through this context's `ctx.llm.stream()`. */
    llmStreamCall: true
  }
  | {
    /** Optional complete output from an unmarked template, remote, or other summarizer. */
    rawOutput?: ContentBlock[]
    /** An unmarked summary does not identify a call through this context's LLM seam. */
    llmStreamCall?: never
  }
)
```

ContentBlock · TokenUsage

packages/compaction/compaction/src/types.ts:34

feedback/*

feedback/record — log-only

```
/**
 * One recorded human remark about this session. Log-only and independent
 * of its trigger; it never enters model context or derived history.
 */
'feedback/record': { text: string }
```

packages/feedback/command-feedback/src/index.ts:62

goal/*

goal/change — log-only

```
/**
 * Complete post-mutation goal state or clear tombstone.
 */
'goal/change': GoalChangeMeta
```

packages/goal/goal/src/domain.ts:66

hook/*

hook/invoked — log-only

```
/**
 * A hook command was invoked at a hook point — a log-only record (like
 * `compaction/*`; NOT a {@link SurfaceEventType}, carries no `surfaceOp`).
 * `dialect` is the bridge that ran it (`claude`/`codex`), `point`
 * the hook point (`PreToolUse`, `Stop`, ...), `matcher` the matcher-group
 * pattern that selected it (absent for match-all), `handlerId` a stable id
 * for the command (so an invoked/result pair correlates). `turn` is the open
 * turn the invocation lives inside.
```

```
 */
'hook/invoked': {
  turn: number
  point: string
  dialect: HookDialect
  matcher?: string
  handlerId: string
}
```

packages/hooks/hook-protocol/src/types.ts:19

`hook/result` — log-only

```
/**
 * Log-only outcome paired to `hook/invoked` by `handlerId`. Decision is the
 * parsed permission result, `stop` for `continue:false`, or `pass`; exit code
 * may be absent, stderr is bounded, and duration is wall-clock runtime.
 */
'hook/result': {
  turn: number
  point: string
  handlerId: string
  decision: string
  exitCode?: number
  stderrSummary?: string
  durationMs: number
}
```

packages/hooks/hook-protocol/src/types.ts:31

`llm/*`

`llm/retry` — log-only

```
/** Durable, non-surface record of one provider-routed retry scheduled after a failed request attempt. */
'llm/retry': LlmRetryEventData
```

packages/llm/llm-retry/src/types.ts:9

llm/retry-started — log-only

```
/** Durable transition written after a retry wait succeeds and before the next request attempt starts. */
'llm/retry-started': LlmRetryStartedEventData
```

packages/llm/llm-retry/src/types.ts:11

model/*

model/selection — log-only

```
/**
 * Complete validated model selection requested for subsequent prompt
 * assembly. Log-only: it never enters derived model history.
 */
'model/selection': ModelSelection
```

packages/api/session-controller/src/types.ts:41

permission/*

permission/preset — log-only

```
/**
 * Records the selected preset as durable, log-only user intent. The knob
 * events follow in the same turn and control execution; this event stays
 * out of the model transcript and lets the permission projection unit
 * preserve a selection when bundles match.
 */
'permission/preset': { preset: string }
```

packages/interaction/permission-presets/src/index.ts:53

plan/*

plan/mode — log-only

```
/**
 * Whether plan mode is in force from this point on: log-only, non-surface,
 * whole-value replace. The last `plan/mode` wins; a log with none folds to
 * inactive through the projection unit's fold.
 */
'plan/mode': { active: boolean }
```

packages/plan/plan-mode/src/index.ts:46

request/*

request/context — log-only

```
/**
 * Route metadata for the next request, logged only when the route or capacity
 * changes. It does not participate in request reconstruction or header equality.
 */
'request/context': RequestContext
```

packages/core/session/src/types.ts:339

request/header — log-only

```
/**
 * Full header for the next request, appended inside its step before dispatch.
 * It is log-only; the latest snapshot reconstructs the request header.
 */
'request/header': {
  header: EpochHeader
  reason: RequestHeaderReason
  /** A changed header also begins a distinct model-message series. */
  startsSeries?: true
}
```

packages/core/session/src/types.ts:329

sandbox/*

`sandbox/mode` — log-only

```
/**
 * The session's sandbox mode was switched — log-only (like `approval/*`;
 * NOT a surface event, carries no `surfaceOp`): durable and replayable,
 * never in the model transcript. The LAST such event is the session's
 * override (folded by the sandboxMode projection unit). `source: 'delegation'` marks
 * an override seeded into a child; an absent source is a runtime switch.
 */
'sandbox/mode': {
  mode: SandboxMode
  /** Marks an override seeded into a child at delegation. */
  source?: 'delegation'
}
```

packages/sandbox/sandbox-policy/src/session-mode.ts:33

`schedule/*`

`schedule/change` — log-only

```
/**
 * Versioned Schedule mutation. The owning package validates the complete
 * session-local transition stream before accepting a candidate event.
 */
'schedule/change': ScheduleChange
```

ScheduleChange

packages/schedule/schedule/src/types.ts:219

`session/*`

`session/end-seed` — log-only

```
/**
 * Marks the end of a constructor seed. Events before it have smaller seq
```

```
 * values and came from the seed (resume, fork, or replay); this lifecycle
 * produced none of them. This log-only event is the durable projection of
 * {@link Session.firstLiveSeq}. Its payload is empty — position and `time`
 * carry the meaning.
 *
 * Locate the LAST one in stored history. A seed already ending in one is not
 * re-marked, so reopening an untouched session does not grow its log per
 * pickup and the event need not be at the current `firstLiveSeq`.
 *
 * `Session`'s constructor is the only legitimate writer. The invariant
 * companion deliberately constrains nothing here, so a plugin appending one
 * would silently classify every live bracket before it as seed history.
 *
 * An owner of a standalone open/close bracket (`compaction/start` …
 * `compaction/end`) reads it because seed history and live work are otherwise
 * byte-identical: an unmatched opening marker before this event belongs to
 * an ended lifecycle, whatever ended it. NOT a liveness signal about other
 * writers — a concurrently live session holds its own boundary elsewhere,
 * so tolerating concurrent writers needs a signal beyond the log.
 */
'session/end-seed': Record<string, never>
```

packages/core/session/src/types.ts:362

session/title — log-only

```
/**
 * Latest-wins session title snapshot. Log-only: it never enters the model
 * surface or derived history.
 */
'session/title': SessionTitleEventData
```

SessionTitleEventData

packages/session/session-title/src/index.ts:77

session/title-llm-request — log-only

```
/** Log-only pre-dispatch record of one session-title model request. */
'session/title-llm-request': SessionTitleLlmRequestEventData
```

SessionTitleLlmRequestEventData

packages/session/session-title-llm/src/index.ts:45

session-log-deepseek/*

session-log-deepseek/delivery-accepted — log-only

```
/** Records that the configured endpoint accepted one delivery through `throughSeq`. */
'session-log-deepseek/delivery-accepted': {
  /** Session identity the accepted delivery carried; inherited fork markers retain the parent's id. */
  sessionId: import('@deepseek-ai/dsh-session/types').SessionId
  /** Last canonical event included in the accepted request. */
  throughSeq: import('@deepseek-ai/dsh-session/types').SessionSeq
}
```

packages/session/session-log-deepseek/src/types.ts:57

step/*

step/end — log-only

```
/** Closes step `step` of turn `turn`. */
'step/end': { turn: number; step: number }
```

packages/core/session/src/types.ts:279

step/start — log-only

```
/** Opens step `step` of turn `turn` — one model call plus the tool executions it requested. */
'step/start': { turn: number; step: number }
```

packages/core/session/src/types.ts:277

subagent/*

subagent/descriptor — log-only

```
/**
 * Durable identity and lifecycle mode of a session-backed subagent child,
 * appended once by the establishing provider inside the child's initial
 * turn, before its first request. Continuable records also carry their
 * resumable composition. Log-only: it carries no `surfaceOp`, never enters
 * model history, and survives compaction.
 */
'subagent/descriptor': SubagentDescriptorData
```

packages/subagent/subagent/src/descriptor.ts:38

subagent/model-selection-policy — □□□

```
/**
 * Records that this session's delegation tool exposes child provider,
 * model, and reasoning-effort selection. Appended before the first model
 * request; absence means the fixed-route definition. Log-only: it carries
 * no `surfaceOp` and never enters model history.
 */
'subagent/model-selection-policy': {
  /** Exact routes this Session may select explicitly for a child. */
  allowedModels: AllowedModelRoute[]
}
```

packages/subagent/tool-subagent/src/model-selection-state.ts:17

team/*

team/member — log-only

```
/** Whole teammate lifecycle value, stored only in the Team Lead Session. */
'team/member': { version: 1; teamId: TeamId; member: TeamMemberSnapshot }
```

TeamId · TeamMemberSnapshot

packages/experimental/agent-team/src/types.ts:206

team/message/delivered — log-only

```
/** Durable acknowledgement that the target Session recorded the message. */
'team/message/delivered': {
  version: 1
  teamId: TeamId
  messageId: TeamMessageId
  targetId: SessionId
}
```

TeamId · TeamMessageId

packages/experimental/agent-team/src/types.ts:212

team/message/queued — log-only

```
/** Durable mailbox enqueue, stored before delivery is attempted. */
'team/message/queued': { version: 1; teamId: TeamId; message: TeamMessageSnapshot }
```

TeamId · TeamMessageSnapshot

packages/experimental/agent-team/src/types.ts:210

team/task — log-only

```
/** Whole shared-task value, stored only in the Team Lead Session. */
'team/task': { version: 1; teamId: TeamId; task: TeamTaskSnapshot }
```

TeamId · TeamTaskSnapshot

packages/experimental/agent-team/src/types.ts:208

todo/*

`todo/write` — log-only

```
/** Whole-list snapshot; latest write wins on replay. Log-only UI state; never derived history. */
'todo/write': { todos: TodoItem[] }
```

TodoItem

packages/todo/tool-todo/src/types.ts:31

tool/*

`tool/call` — log-only

```
/**
 * The model requested one tool invocation: `name` with the raw `arguments`
 * JSON string exactly as the model produced it (unparsed). `callId` pairs the
 * call with its `tool/result`.
 */
'tool/call': { turn: number; step: number; callId: ToolCallId; name: string; arguments: string }
```

ToolCallId

packages/core/session/src/types.ts:306

`tool/code-dispatch` — log-only

```
/**
 * One bridged sub-dispatch SETTLING: the pairing ids (matching the
 * `tool/code-dispatch-start` with the same `subCallId`), the tool `name`
 * with the same JSON-normalized `arguments`, and the sub-call's complete
 * model-facing outcome in `tool/result`'s own vocabulary
 * (`content` + `isError`), so UIs render a sub-call through the exact
 * code path that renders a native call. Every started sub-call settles
 * with exactly one of these (abort included: the aborted pipeline result
 * is an `isError` outcome).
 * Log-only: `deriveMessages()` ignores it, so sub-calls never re-enter
 * model context; persistence and UIs get every call. Appended inside the
 * parent `run_code`'s execution (the bridge drains in-flight dispatches
 * before returning), so its execution-enclosure relation holds by
```

```
 * construction.
 */
'tool/code-dispatch': PtcDispatchEventData
```

packages/core/tools/src/types.ts:56

`tool/code-dispatch-start` — log-only

```
/**
 * One sub-dispatch STARTING inside a `run_code` program: the parent
 * `run_code` call id, the deterministic sub-call id (`<parent>:code:<n>`,
 * numbered in submission order), and the tool `name` with its
 * JSON-normalized `arguments` — the exact value dispatched, normalized
 * BEFORE dispatch, so this append can never fail on payload shape.
 * Appended when the scheduler actually starts the call (not at
 * submission), so a start means the tool body pipeline was entered; a
 * call abandoned in the queue logs nothing. Log-only: `deriveMessages()`
 * ignores it; UIs use it for live per-sub-call running state and pair it
 * with `tool/code-dispatch` by `subCallId` (timing = the two events'
 * `time` fields).
 */
'tool/code-dispatch-start': PtcDispatchStartEventData
```

packages/core/tools/src/types.ts:40

`tool/result` — surface

```
/**
 * A completed tool call's model-facing result, optional internal failure
 * identity, and optional tool-private `meta` presentation payload. `meta` is
 * opaque to the core (the producing tool owns its shape and reads it back in
 * `presentResult`) but MUST be JSON-serializable: `Session.append`
 * runtime-validates all event data with `isJsonValue`, so a non-serializable
 * `meta` is rejected at the source, and the durable log reproduces the
 * identical card on replay. Absent
 * unless the tool attaches one (e.g. `dsh-tool-fs` carries its result-time
 * contextual diff here).
 */
'tool/result': {
```

```
  turn: number
  step: number
  message: ToolResultMessage
  error?: { name: string; code: string }
  meta?: JsonValue
}
```

packages/core/session/src/types.ts:318

tool-workflow/*

tool-workflow/agent-end — log-only

```
/**
 * Records one member settlement.
 * @param data - run identity, paired member sequence, and outcome.
 */
'tool-workflow/agent-end': ToolWorkflowAgentEndData
```

packages/workflow/tool-workflow/src/types.ts:57

tool-workflow/agent-start — log-only

```
/**
 * Records one published workflow member.
 * @param data - run identity, member sequence, display identity, and child Session.
 */
'tool-workflow/agent-start': ToolWorkflowAgentStartData
```

packages/workflow/tool-workflow/src/types.ts:52

tool-workflow/run-end — log-only

```
/**
 * Closes one workflow record after cleanup.
 * @param data - stable run identity and terminal reason.
 */
'tool-workflow/run-end': ToolWorkflowRunEndData
```

packages/workflow/tool-workflow/src/types.ts:62

`tool-workflow/run-start` — log-only

```
/**
 * Opens one top-level workflow record.
 * @param data - stable run identity and display name.
 */
'tool-workflow/run-start': ToolWorkflowRunStartData
```

packages/workflow/tool-workflow/src/types.ts:47

`turn/*`

`turn/end` — log-only

```
/**
 * Closes turn `turn` with the {@link TurnEndReason} that ended it. A turn
 * with no entered step has no `step/start` or `step/end`. The loop does not await a
 * flush at turn boundaries: `dsh-session-checkpoint-policy` owns the
 * per-request durability checkpoint, and consumers that read storage after
 * `whenIdle()` flush themselves. Success commits the turn; rejection is
 * reported live and does not prevent later work.
 */
'turn/end': { turn: number; reason: TurnEndReason }
```

TurnEndReason

packages/core/session/src/types.ts:275

`turn/start` — log-only

```
/**
 * Opens turn `turn` before the loop claims queued input or runs pre-step.
 * Rejection, empty input, cancellation, or failure may close it with no
 * step; otherwise the following identified `user/message` event or batch
 * records the messages entering the step.
 */
```

```
'turn/start': { turn: number }
```

packages/core/session/src/types.ts:266

user/*

user/message — surface

```
/**
 * A user-role message on the model-visible surface: a direct human prompt
 * (the queued message claimed for this turn), a synthetic `agent.inject()`
 * context (file-change notices, subdir AGENTS.md, skill content, cron
 * notifications, …), or an entered goal continuation round. All three
 * project their `content` verbatim; `source` tells them apart.
 */
'user/message': UserMessage
```

packages/core/session/src/types.ts:287

web/*

web/deepseek-search-llm-request — log-only

```
/** Secret-free auxiliary DeepSeek search request recorded before dispatch. */
'web/deepseek-search-llm-request': DeepSeekSearchLlmRequest
```

packages/web/web-search-deepseek/src/provider.ts:83

版本号 #1
由一说春秋 创建于 3 九月 2026 17:28:49
由一说春秋 更新于 3 九月 2026 17:36:27