DeepSeek Harness Handbook

dsh-handbook · Complete English Edition

From zero to one with DeepSeek's open-source agent runtime — the beginner's encyclopedia: install, use, develop plugins, tune, secure, and budget.

14 Chapters + Appendix (Glossary / Packages / Benchmark)

Synced from docs/ · dsh 0.1.0-rc.6 · 2026-08-14

https://github.com/Electricitysheep/dsh-handbook

Contents

Chapter 1: Understanding DeepSeek Harness

*Goal: build a complete mental model of DeepSeek Harness (dsh) from absolute zero — **what it is, why it matters, how it differs from mainstream agents, and when to use it**. No prior knowledge required.*

1.1 Three intuitions first

① What is dsh? — "the LEGO baseplate for agents."

LEGO gives you the baseplate and standard bricks (runtime + core plugins); you assemble freely (add plugins, swap UIs, change behavior). Claude Code, by contrast, is "a finished car" — great to drive, but modifying the engine means asking the vendor.

② Why "harness"? — the engineering layer around a model.

A model (DeepSeek V4) only "replies with text." To make it work in your repository (read files, run commands, edit code, loop), you need an engineering layer on top: session management, tool calling, context control, error recovery. **That layer is a harness.** dsh is DeepSeek open-sourcing that layer.

③ Why open-source it now? — agents entered the "programmable era".

2025 was model-capability competition; 2026 is agent-engineering competition. Open-sourcing the harness is the strategic move to make "how to organize agents" an open ecosystem — like Android did for phones.

1.2 Official facts

One-liner: DeepSeek Harness (`dsh`) is DeepSeek's open-source agent runtime with an "everything is a plugin" architecture, built on the Cordis plugin container.

Fact	Value
Open-sourced	2026-08-13
License	MIT
Language	TypeScript (Node.js ≥ 22)
Version	`0.1.0-rc.x` (rc.6; fast iteration, breaking changes expected)
Runtime	Cordis
Built-in profiles	`web` + `headless`

1.3 How "everything is a plugin" works

Layered view

```
┌──────────────────────────────────────────────┐
│  Your profile (bootable form)                │
│  = bundle stack + your patch layer           │
```

```
│  · dsh web (Web UI)                              │
│  · dsh headless (CLI)                            │
│  · your custom profile (TUI/desktop/bot…)        │
├──────────────────────────────────────────────────┤
│  Capability layer (each = a plugin)              │
│  · llm · tools · session · client · settings     │
│  … (60+ official packages)                       │
├──────────────────────────────────────────────────┤
│  Cordis container: loading, DI, events, lifecycle │
└──────────────────────────────────────────────────┘
```

Three concepts you must know

① **Profile (bootable form)** — a directory `$DSH_HOME/profiles/<name>/` with `package.json` (plugin deps + manifest) and `cordis.patch.yml` (your patch layer). Load order: built-in bundles → profile patch → global patch → `--patch` overlays.

② **host half / client half** — one npm package can carry both: the host half (`apply(ctx)` on Node: tools, services, events) and the client half (browser UI, declared via `dsh.client` in package.json).

③ **Extension points** — official principle: *"Plugins, not loop changes."* Use hooks, don't fork the core. Common ones: `agent/request` waterfall (change request config per step), `conversationEvents.register`, `ctx.slots.inject` (inject UI), `settings` service.

1.4 dsh vs mainstream agents

Capability matrix

Dimension	dsh	Claude Code	OpenAI Codex	OpenCode	Gemini CLI	Kimi CLI
Open source	✅ MIT	❌	❌	✅ MIT	❌	❌
Model binding	model-agnostic	Claude	GPT	any	Gemini	Kimi
Official runtime	✅ + plugin ecosystem	product	product	client only	product	product
Plugin system	**first-class, 60+ official pkgs**	config/hooks	config	config	none	none
Custom UI	✅ (client half)	❌	❌	partial	❌	❌
Automation/ CI	✅ headless	✅	✅	✅	✅	✅
TUI	plugin-provided	✅ built-in	✅ built-in	✅ built-in	✅	✅
Ecosystem stage	day-zero (2026-08-13)	mature	mature	mature	mature	early
Best for	**deep customizatio n +**	out-of-box	out-of-box	OpenCode users	Google	Kimi

	ecosystem					

Case: same task, different doors

Task: "Find every call of function X in the repo and change one argument."

Agent	How you do it
dsh (web)	`dsh web` → type → model uses Grep/Read/Edit tools; add plugin sidebars (Git panel, speed-up plugin)
dsh (headless)	`dsh --profile headless "task"` → prints result, exits — **CI-friendly**
Claude Code / Codex / OpenCode / Kimi	open the TUI → type → model does it

The difference: same sentence, but dsh gives you a choice dimension — swap the UI and toolchain. Others ship a fixed interface.

Case: real workloads (our measurements, 2026-08-13)

Scenario	dsh measured	Note
Simple file create	cold start ~110s (first round, context injection) → hot cache ~1s	reasoning effort is the main variable
50-step tool chain	LLM 10m+, tool calls 9m+	per-step thinking accumulates — **that's the speed plugin's value**
Plugin dev	zero to runnable plugin: 1 day (tests + live verify)	clear extension points
vs Claude Code same task	dsh + V4-Flash ≈ 1/10–1/30 of Claude cost	price dimension favors dsh ecosystem

1.5 When to use dsh

✅ **Adopt now**: model-agnostic + UI-optional + behavior-mutable base; be an early ecosystem contributor; run agents on servers/CI; cost-sensitive.

⏸ **Wait**: you need an out-of-the-box coding assistant; can't accept rc breaking changes; deeply dependent on a vendor's exclusive model feature.

1.6 FAQ

- **Is dsh a model?** No. It's a runtime; models plug in via the `llm` plugin (DeepSeek V4 native, OpenAI-compatible others possible).
- **dsh vs OpenCode?** Both open-source agent clients; OpenCode is "client + config", dsh is "runtime + official plugin ecosystem" (official bundles, 60+ packages).
- **Do I need TypeScript?** To *use* it, no. To *write plugins* (Ch.4), basic TS — with full code in the handbook.

- **Is it stable?** rc stage; use for ecosystem/tinkering now, production core after 0.1.0.
- **Why learn dsh now?** Day-zero ecosystem + official encouragement + Chinese-tutorial vacuum — first-mover window.

Chapter 2: 5-Minute Quickstart

*Goal: **follow along and get it running.** Every command shows its expected output and common errors. Open a terminal and go.*

2.1 Prerequisites (30-second check)

Need	Check	Pass
Node.js ≥ 22	`node --version`	v22.x+ (24 recommended)
npm	`npm --version`	prints a version
Network	npm registry reachable	can install packages
(optional) DeepSeek API key	https://platform.deepseek.com	for real conversations

dsh starts without a key (UI opens), but conversations need one. Examples assume it's configured.

2.2 Install (two ways)

One-liner (recommended for new users)

```
npx -y @deepseek-ai/dsh --version
```

First run downloads dsh (large package, 40+ plugin modules, 1-3 min). You're good when you see:

```
0.1.0-rc.6
```

Global install (recommended for frequent use)

```
npm install -g @deepseek-ai/dsh
dsh --version
```

2.3 Mode 1: Web UI (`dsh web`)

Start

```
dsh web
```

Expected output:

```
dsh web: http://127.0.0.1:3080
```

Open http://127.0.0.1:3080 in your browser.

UI map (see screenshot)

Area	What's there
Left	session list / workspace switch / new session
Center	chat: input box, model selector (`DeepSeek V4 Flash`), reasoning level (`High`)
Right/bottom	plugin sidebar (empty by default; appears after installing community plugins)
Top-right	Session log / Trajectory

First conversation

1. Click **New session**
2. Type: `Hello, introduce yourself in one sentence`
3. Press Enter

Expected reply (roughly):

> *Hello! I'm a DeepSeek-powered AI coding assistant...*

Models & reasoning effort

Click the model selector next to the input:

Model	Positioning
`deepseek-v4-flash` (default)	value: fast, cheap, enough for daily work
`deepseek-v4-pro`	flagship: stronger, more expensive/slower

Reasoning effort (three steps since 2026-08-13):

Level	Speed	Quality	Suggested use
`low`	fastest	adequate	simple/deterministic rounds, batch, cheap chain steps
`high` (default)	medium	good	daily agent tasks
`max`	slowest	best	hard reasoning, long planning

> 💡 ***Key performance insight**: the model re-thinks **before every tool call**. Measured: a "create file" task spends ~90% of wall-clock in thinking; a 50-step tool chain accumulates minutes. **Lowering the reasoning effort is the highest-leverage speedup** (Ch.6 + the example speed-up plugin).*

2.4 Mode 2: Headless (one-shot tasks, scripts/CI)

```
dsh --profile headless "Hello, introduce yourself in one sentence"
```

Expected output (prints result, process exits):

```
Hello! I'm a DeepSeek-powered AI coding assistant...
```

Headless value:

- **Automation**: CI, servers, cron
- **Script-friendly**: non-zero exit = failure; output pipes
- **Isolation**: fresh session per call (`--resume` restores; see `dsh --profile headless --help`)

Example: daily digest script:

```
dsh --profile headless "Read today's git log in the workspace and write a Chinese daily
summary" > daily-report.md
echo "exit=$?"
```

2.5 Your first plugin: a Git panel for web

dsh's sidebar is empty by default — install community plugin `dsh-better-sidebar` to feel "everything is a plugin" (mechanics in Ch.3; here just get it running):

```
# 1. locate your web profile
#    Windows: %USERPROFILE%\.dsh\profiles\web
#    macOS/Linux: ~/.dsh/profiles/web

# 2. add one line to package.json dependencies
#    "dsh-better-sidebar": "link:C:\\path\\to\\DSH-better-sidebar"

# 3. add to cordis.patch.yml
#    - insert:
#        - id: better-sidebar
#          name: dsh-better-sidebar

# 4. install & restart
cd ~/.dsh/profiles/web && pnpm install
dsh web
```

After restart, the sidebar gains file manager / terminal / **Git panel** / browser tabs.

*The panel's fetch/pull/push buttons are a community PR (Ch.5) — **this is how the plugin ecosystem works.***

2.6 Config & paths

First run creates:

```
~/.dsh/
├── settings.yaml          # global settings (model, reasoning effort)
├── profiles/              # profile dirs
│   └── web/
│       ├── package.json      # plugin deps + manifest
```

```
│       └── cordis.patch.yml  # your patch layer
├── sessions/              # session data
└── storages/              # persistence
```

`settings.yaml` example:

```
agent-default-model:
  model: deepseek-v4-flash
  reasoningEffort: high
```

2.7 Command cheatsheet

Command	Purpose
`dsh web`	start Web UI (alias `dsh --profile web`)
`dsh --profile headless "task"`	one-shot task, print result, exit
`dsh plugin --profile <name> add <pkg>`	add a plugin to a profile
`dsh --dump-config`	print composed config tree
`dsh --profile tui`	TUI mode (plugin-provided; not built-in)
`dsh --version`	version

2.8 Troubleshooting

Symptom	Cause & fix
`dsh: profile "tui" does not exist`	tui needs a plugin (`dsh plugin --profile tui add <pkg>`)
npx very slow	first-run package size; `npm i -g` helps
browser can't reach 3080	port busy: `netstat -ano | findstr 3080` → kill PID
model not responding	check `~/.dsh/settings.yaml` + API key
plugin install 404	**rc.1 dependency break**: use `^0.1.0-rc.6` line (Ch.3 pitfall #1)
behavior changed after upgrade	rc-stage breaking changes; check changelog

Chapter 3: Profiles & the Plugin System

> ***Goal of this chapter:*** *Understand dsh's customizable skeleton, how profiles are organized, how plugins are mounted, and what the host/client dual halves are.* ***This is the watershed moment from "user" to "developer."***

TL;DR (30-second version)

1. **Profile = a launchable form factor**: a `~/.dsh/profiles/<name>/` directory composed of `package.json` (plugin manifest) + `cordis.patch.yml` (patch layer).
2. **Mounting a plugin takes two steps**: add a dependency in `package.json` + add a mount line in `cordis.patch.yml`, then `pnpm install` and restart.
3. **Host half runs in Node, client half runs in the browser**: one npm package carries both faces via `exports["."]` and `exports["./client"]`.
4. **To change behavior, look for extension points first, don't fork the core**: `agent/request` waterfall, `conversationEvents`, `ctx.slots.inject`, and the `settings` service are the four most common hooks.
5. **Three rc-stage pitfalls**: use the `^0.1.0-rc.6` dependency line, always `await next()`, and client tests need the dsh runtime.

3.1 Profile: A Launchable Configuration Stack

dsh uses **profiles** to represent "a launchable form factor." Two are built in; the rest are created via plugins:

Profile	Purpose	Command
`web`	Web UI (conversation + sidebar + tools)	`dsh web`
`headless`	One-shot CLI tasks	`dsh --profile headless "task"`
`tui` (plugin required)	Terminal UI	`dsh --profile tui` (not built in; requires a plugin)

A profile directory looks like this (`~/.dsh/profiles/<name>/`):

```
profiles/web/
├── package.json        # Plugin dependencies + dsh.profile manifest (bundles order)
├── cordis.patch.yml    # Your patch layer: mount/override plugin declarations
├── cordis.yml          # (Generated) Composite configuration
├── pnpm-workspace.yaml
└── node_modules/
```

Loading order (from official docs): Built-in bundles (`dsh-base` → `dsh-web-app`) → profile's `cordis.patch.yml` → user-level `~/.dsh/cordis.patch.yml` → `--patch` overlay.

3.2 Mounting a Plugin: Two Changes

Here's how to mount the community plugin `dsh-better-sidebar` (a real operation):

① Add the dependency in `package.json` (using `link:` to point to local source, or an npm package name):

```
{
  "dependencies": {
    "dsh-better-sidebar": "link:C:\\path\\to\\DSH-better-sidebar"
  }
}
```

② Add the mount line in `cordis.patch.yml`:

```
- insert:
    - id: better-sidebar
      name: dsh-better-sidebar
```

③ Install and restart:

```
cd ~/.dsh/profiles/web && pnpm install
dsh web   # Takes effect after restart
```

> 💡 *Plugins are **isolated per session** by default (e.g. `better-sidebar`'s layout/tabs persist per session). Mounting is profile-level, but state is session-level.*

3.3 Host Half & Client Half: One Package, Two Faces

A plugin can carry two runtime halves simultaneously:

Half	Runs In	Responsibility	Example
Host half	Node process	Tools, services, events, filesystem, processes	`apply(ctx)` registers tools/services
Client half	Browser (web profile)	UI, interaction, DOM	`dsh.client` declaration in `package.json` + `src/client/`

How to declare the client half in `package.json`:

```
{
  "dsh": {
    "client": {
      "inject": ["@deepseek-ai/dsh-client-runtime", "@deepseek-ai/dsh-client-locale"],
      "platform": "web"
    }
  },
```

```
  "exports": {
    ".": { "types": "./lib/types/index.d.ts", "default": "./lib/index.js" },
    "./client": { "types": "./lib/types/client/index.d.ts", "default": "./lib/client.js" }
  }
}
```

- Host half: `exports["."]` → `apply(ctx)` (the cordis plugin body)
- Client half: `exports["./client"]` → `apply(ctx)` on the browser side
- `inject`: Declares required services (cordis dependency injection)

3.4 Extension Points: Look for Hooks First, Don't Fork the Core

Official principle (stated in CONTRIBUTING/AGENTS.md): **"Plugins, not loop changes: new behavior goes on documented extension points."** The most common mistake newcomers make is modifying the core loop. The correct approach is to use extension points.

Confirmed commonly-used extension points (each explored hands-on in later chapters):

Extension Point	Location	Purpose
`agent/request` waterfall	`agent-loop`	**Modify config before every model request** (provider/model/reasoningEffort/tools). This is what the speed-up plugin example uses.
`agent/request-error`	`agent-loop`	Intervene on request failure (the official compaction plugin uses this for context-window overflow recovery)
`conversationEvents.register`	Client runtime	Subscribe to / inject conversation events (tool/call, turn/start, etc.)
`ctx.slots.inject`	Client UI slots	Inject UI into interface slots (e.g. turnTail to display artifact file lines)
`settings` service	dsh-settings	Register user-configurable namespaces (settings page auto-renders them)
`ctx.provide` / `ctx.get`	cordis	Provide services across plugins

3.5 Common Pitfalls (Battle-Tested)

Pitfall	Symptom	Fix
rc.1 broken dependency chain	`pnpm install` reports `@deepseek-ai/dsh-type-meta@0.0.1-rc.1` 404	Several packages from the rc.1 era were never published; upgrade to the `^0.1.0-rc.6` line
Plugin missing `main`	`dsh: No "exports" main defined`	The host plugin's `package.json` must expose a `.` entry (`"main": "src/index.ts"` can be loaded directly by tsx)
Event handler forgets `await`	Request loses provider/model and	`next()` in `agent/request` returns

`next()`	errors out	a **Promise**; you must await it before spreading
Type error: `'agent/request' is not assignable to keyof Events`	npm types don't re-export the official type augmentations	Use a relaxed signature (`ctx.on as unknown as ...`) at the boundary
Client package depends on `window.__ModuleLoader__`	jsdom can't run client tests directly	Component-level tests need the dsh web runtime (run in official CI)

Hands-on exercises

1. **Locate your profile**: find the web profile directory on your machine. Read `package.json` and `cordis.patch.yml`. Identify which plugins are currently mounted.
2. **Mount a plugin**: follow Section 3.2 to mount `dsh-better-sidebar` (or any community plugin). Verify it appears after restart.
3. **Trace the host/client split**: pick a plugin that has both halves. Find the `dsh.client` declaration in its `package.json` and the `apply(ctx)` function in its entry file.
4. **Find an extension point**: read the `agent/request` waterfall description above. In the dsh source, search for where this waterfall is called. What other waterfalls or hooks can you spot?
5. **Pitfall reproduction**: deliberately forget to `await next()` in a test plugin. Observe the error. Then fix it and confirm the config flows through.
6. **Think**: why does dsh separate "profile" from "plugin"? What would break if plugins were global instead of profile-scoped?

FAQ

- **Q: What's the difference between a profile and a plugin?** A profile is a *launchable form factor* (a directory with config + patch). A plugin is a *capability unit* mounted into a profile. You can think of a profile as a "container" and plugins as "modules inside it."
- **Q: Do I need to create a new profile for every project?** No. Most users stick with `web` for interactive work and `headless` for scripting. Custom profiles are for advanced use cases (TUI, desktop, bots).
- **Q: Why does `pnpm install` fail with 404 errors?** The rc.1 dependency chain is broken. Use the `^0.1.0-rc.6` line in `package.json` (see Section 3.5).
- **Q: Can I use a plugin without modifying `cordis.patch.yml`?** No. Adding the dependency alone doesn't activate the plugin. You must declare it in the patch layer for Cordis to load it.
- **Q: Why can't I run client-half tests with jsdom?** The client half depends on `window.__ModuleLoader__`, which is bootstrapped by the dsh web runtime. Component tests need the official CI environment.

Next chapter: Chapter 4: Plugin Development, Hands-On (planned) — Write your first host plugin from scratch.

Chapter 4: Plugin Development, Hands-On

> ***Goal of this chapter:*** *Build a* ***real, working host plugin*** *from scratch, one that automatically adjusts reasoning effort via the* `agent/request` *extension point. This is a full breakdown of an example speed-up plugin. All code is runnable and testable.*

TL;DR (30-second version)

1. **Core problem**: dsh re-thinks before every tool call; in a 50-step task, 90%+ of wall-clock time is thinking. Lowering the reasoning effort is the fastest speedup.
2. **Three-step architecture**: pure function (decision logic, zero deps, unit-testable) → plugin body (hook into the waterfall) → live verification (logs prove the injection happened).
3. **`agent/request` waterfall**: fires before every model request; listener return values flow to the next listener, enabling "keep original config + override one field."
4. **`next()` is a Promise; you must `await` it**: spreading without awaiting yields an empty object, dropping the provider/model and causing errors.
5. **Development discipline**: find the extension point first (90% of behaviors have official hooks), extract logic into pure functions, never skip live verification.

4.1 What We're Building

Problem: Before every tool call, the dsh model re-thinks from scratch (`reasoning_effort`). In a 50-step tool-chain task, "thinking" accounts for 90%+ of wall-clock time.

Solution: A host plugin that listens on the `agent/request` waterfall and downgrades `reasoning_effort` from `high` to `low` for simple turns, based on the most recent tool calls.

4.2 Project Skeleton

```
dsh-speed-plugin/
├── package.json          # Host plugin declaration
├── tsconfig.json
├── src/
│   ├── effort-decision.ts  # Pure function: decision logic (zero deps, unit-testable)
│   └── index.ts            # apply(ctx): hooks into the extension point
└── tests/
    └── effort-decision.spec.ts
```

Key fields in `package.json`:

```
{
  "name": "dsh-speed-plugin",
  "type": "module",
```

```
  "main": "src/index.ts",
  "exports": {
    ".": { "types": "./src/index.ts", "default": "./src/index.ts" }
  },
  "peerDependencies": {
    "@deepseek-ai/cordis": "^4.0.1",
    "@deepseek-ai/dsh-agent": "^0.1.0-rc.6"
  }
}
```

⚠ *Always use the `^0.1.0-rc.6` dependency line. The rc.1 npm dependency chain is broken (see Chapter 3 pitfalls).*

4.3 Pure Function: Decision Logic (Zero Dependencies, Unit-Testable)

`src/effort-decision.ts`:

```
export type EffortId = 'low' | 'high' | 'max'

export interface ToolCallSample {
  name: string      // Tool name, e.g. 'write', 'read', 'bash'
  argsSize: number  // Argument size (character count)
}

export interface EffortDecisionInput {
  recentCalls: readonly ToolCallSample[]
  selected: EffortId      // User's baseline effort
  allowDowngrade: boolean
  allowUpgrade: boolean
}

const SIMPLE_TOOL_RE = /^(fs|bash|terminal|read|write|grep|glob|edit|ls|cat|rm|cp|touch|
mkdir|pwd)/i
const HEAVY_ARGS = 800

export function decideEffort(input: EffortDecisionInput): EffortId {
  const { recentCalls, selected, allowDowngrade, allowUpgrade } = input
  if (recentCalls.length === 0) return selected   // Fresh prompt: keep baseline

  const ratio = recentCalls.filter(c =>
    SIMPLE_TOOL_RE.test(c.name) && c.argsSize < HEAVY_ARGS,
  ).length / recentCalls.length
  const heaviest = recentCalls.reduce((m, c) => Math.max(m, c.argsSize), 0)

  if (ratio >= 0.75 && allowDowngrade) return 'low'
  if (heaviest >= HEAVY_ARGS * 4 && allowUpgrade) return 'max'
  if (ratio < 0.75) return allowUpgrade ? 'high' : selected
  return selected
}
```

Why extract a pure function: The decision logic is decoupled from the dsh runtime. Unit tests have zero dependencies, run in milliseconds, and cover every branch. Live verification only needs to confirm "did the injection actually happen."

4.4 Plugin Body: Hooking into the `agent/request` Waterfall

`src/index.ts`:

```
import type { Context } from '@deepseek-ai/cordis'
import { decideEffort, type ToolCallSample } from './effort-decision.ts'

export interface SpeedPluginConfig {
  enabled: boolean
  allowDowngrade: boolean
  allowUpgrade: boolean
  baseline: 'low' | 'high' | 'max'
}

export const DEFAULT_CONFIG: SpeedPluginConfig = {
  enabled: true, allowDowngrade: true, allowUpgrade: false, baseline: 'high',
}

const WINDOW = 8

function recentToolCalls(agent: unknown): ToolCallSample[] {
  const events = (agent as { session?: { events?: readonly unknown[] } }).session?.events ??
[]
  const out: ToolCallSample[] = []
  for (let i = events.length - 1; i >= 0 && out.length < WINDOW; i--) {
    const e = events[i] as { type?: string; data?: { name?: string; arguments?: unknown } } |
undefined
    if (e?.type !== 'tool/call') continue
    out.push({
      name: e.data?.name ?? 'tool',
      argsSize: typeof e.data?.arguments === 'string' ? e.data.arguments.length : 0,
    })
  }
  return out.reverse()
}

export function apply(ctx: Context, config: SpeedPluginConfig = DEFAULT_CONFIG): void {
  if (!config.enabled) return

  // Boundary adaptation: npm package doesn't re-export official event type augmentations,
  // so we relax the signature here (see Chapter 3 pitfalls)
  const on = ctx.on as unknown as (
    event: string,
    handler: (payload: Record<string, unknown>, next: () => unknown) => unknown |
Promise<unknown>,
  ) => void

  on('agent/request', async (payload, next) => {
    const seed = await next() as { reasoningEffort?: unknown }   // ⚠ Must await!
    const calls = recentToolCalls(payload.agent)
    const effort = decideEffort({
      recentCalls: calls,
      selected: config.baseline,
      allowDowngrade: config.allowDowngrade,
      allowUpgrade: config.allowUpgrade,
    })
    console.log(`[speed-plugin] calls=${JSON.stringify(calls)} => reasoningEffort=${effort}`)
    return { ...seed, reasoningEffort: effort }
  })
}
```

Three critical points (all learned the hard way):

1. **`next()` is a Promise:** `await next()` retrieves the current config. Spreading without awaiting yields an empty object, which drops the provider/model and causes errors.
2. **Waterfall semantics:** The listener's **return value** is passed to the next listener and ultimately to the request. Returning `{...seed, reasoningEffort}` means "keep the original config, but override the reasoning effort."
3. **`agent/request` fires every step:** The `agent-loop`'s `buildRequest` runs through this waterfall at every step, so dynamic decisions naturally take effect per-step.

4.5 Testing

Unit tests (pure function, zero dependencies):

```
import { describe, expect, it } from 'vitest'
import { decideEffort } from '../src/effort-decision.ts'

it('downgrades to low for simple tool chains', () => {
  expect(decideEffort({
    recentCalls: [{ name: 'write', argsSize: 40 }],
    selected: 'high', allowDowngrade: true, allowUpgrade: true,
  })).toBe('low')
})
// ... more branches: fresh prompt keeps baseline / downgrade disabled /
//     oversized payload upgrades to max / mixed tools upgrade to high
```

Live verification (critical, proves "the injection actually happens"):

Mount the plugin (Chapter 3 method) → restart `dsh web` → send a file-creation task → watch the dsh process logs:

```
[speed-plugin] agent/request: calls=[]                    => reasoningEffort=high
[speed-plugin] agent/request: calls=[{"name":"write",…}] => reasoningEffort=low
```

First turn has no tool calls → stays at baseline `high`. After detecting the `write` tool → next turn drops to `low`. **The injection pipeline works end to end.**

Full runnable code: combine the code snippets in this chapter to run it.

4.6 Three Development Disciplines for Newcomers

1. **Find the extension point first:** 90% of behaviors you want to change have official hooks (`agent/request`, `settings`, `conversationEvents`, `slots`). Don't fork the core.
2. **Extract logic into pure functions:** Decouple decision/computation logic from dsh. Unit tests run in milliseconds and cover every branch. Live verification only needs to confirm "the injection happened."
3. **Never skip live verification:** Unit tests prove the logic; live logs prove the wiring. Both must pass before you're done.

Hands-on exercises

1. **Read the pure function**: open `src/effort-decision.ts`. Trace the logic: what does `ratio >= 0.75` mean? When does it return `'low'` vs `'max'`?
2. **Write a unit test**: add a test case for "mixed tools with heavy arguments." What effort should it return? Run it.
3. **Break it on purpose**: remove the `await` before `next()` in `src/index.ts`. Run the plugin live. What error do you see? Fix it.
4. **Add a new rule**: extend `decideEffort` to always return `'max'` when the tool name is `'bash'` and the args exceed 2000 characters. Write a test for it.
5. **Live verification**: mount the plugin (Chapter 3 method), restart `dsh web`, send a file-creation task, and watch the logs. Confirm you see `reasoningEffort=low` after the first tool call.
6. **Think**: why extract the decision logic into a pure function instead of putting it directly in the `apply(ctx)` handler? What are the testing and maintenance implications?

FAQ

- **Q: Why not just set `reasoningEffort: low` globally?** Because complex tasks (planning, debugging) benefit from `high` or `max`. The plugin dynamically adjusts per step, giving you the best of both worlds.
- **Q: What if the plugin makes the wrong decision?** The plugin only downgrades when recent tool calls are simple (file ops, grep, etc.). If you're doing complex reasoning, the ratio drops and effort stays high. You can also disable downgrading via config.
- **Q: Can I use this plugin with other models?** The plugin hooks into dsh's `agent/request` waterfall, which is model-agnostic. It should work with any model dsh supports, though the speedup depends on the model's thinking behavior.
- **Q: Why is `next()` a Promise?** The waterfall is async: each listener may need to await the next one. Forgetting `await` breaks the chain and drops config.

- **Q: Where do I find more extension points?** Check `packages/AGENTS.md` in the official repo. The most common ones are listed in Chapter 3, Section 3.4.

Next chapter: Chapter 5: Real-World Cases (planned) — Git panel, HTML draft preview, speed-up plugin.

Chapter 5: What dsh Can Do for You — Application Scenarios

*Goal: step out of the "how to build dsh" lens and look from the **user's** perspective — where dsh genuinely helps in daily work. Every scenario has a copy-paste task example with real measured timing.*

TL;DR (30-second version)

1. **dsh is a general-purpose agent runtime** — not just a coding assistant: data analysis, documentation, automation, and code comprehension all work.
2. **Five core scenarios**: daily coding / data processing / documentation & knowledge / automation & ops / code understanding & refactoring.
3. **One usage pattern**: describe the goal in one sentence + state the acceptance criteria; dsh orchestrates the toolchain itself.
4. **Real speed**: simple tasks in seconds, complex multi-step tasks in 1–5 min (V4-Flash + high effort, measured).
5. **Golden prompt rule**: tell dsh *what "done" looks like* ("run and verify") — more effective than describing the process.

5.1 Scenario overview

Scenario	Typical tasks	dsh tools	Measured
Daily coding	write functions, fix bugs, add tests	write / read / bash / grep	5-bug fix + 49 tests: 94s
Data processing	clean, analyze, visualize, report	read / write / bash / python	clean + visualize: 186s
Docs & knowledge	API docs, tutorials, summaries, translation	read / write	API docs: 1m04s
Automation & ops	batch, CI tasks, log analysis	bash / headless	headless scripting
Code understanding	refactor, review, tech research	grep / glob / read / bash	refactor: 4m37s

All timings from real cases in Chapter 10 (synthetic data); see benchmark.

5.1.1 High cache-hit rate: dsh's hidden cost weapon

The most underrated advantage of the dsh ecosystem. DeepSeek's API **context cache** bills repeated/similar input tokens at a discounted rate. Measured in real dsh sessions: **cache hit rate up to 97%**.

Why dsh hits cache so often:

Reason	Explanation
Session continuity	multi-turn conversations repeat system prompt / skill

	catalog / history
Stable tool schema	every step carries the same tool definitions
Agent workload	multi-step tool chains repeatedly carry the same context

Cost impact (98% cache discount):

Charge	Miss	Cache hit	Discount
Input (Flash)	$0.14/M	**$0.0028/M**	**98%**
Input (Pro)	$0.435/M	$0.003625/M	99%+

Practical meaning: in a long agent session, most input tokens hit cache — a 50-step tool chain with 2.4M input tokens costs far less than the miss-price estimate. For high-frequency / batch / long-session workloads this is an order-of-magnitude advantage.

> *To raise hit rate: keep sessions alive, keep prompt prefixes stable (don't churn system prompt/skill catalog), batch within the same session/prefix.*

5.2 Scenario 1: Daily coding (best entry point)

Typical tasks: write a function/module from a spec; fix bugs (locate → modify → verify); add unit tests.

```
dsh --profile headless "Review buggy_calculator.py: find all bugs, fix them, write complete
tests, run and verify all pass"
# measured: 5 bugs fixed + 49 tests pass (94s)
```

Why dsh fits: coding has the fullest toolchain — read files, write code, run commands, see results. Produced files are directly openable at the end of the conversation.

Prompt tip: always say "run and verify" — dsh closes the loop; "fix my bug" alone stops after editing.

5.3 Scenario 2: Data processing & analysis

```
dsh --profile headless "Analyze sales_data.json for quality issues
(missing/type/duplicate/outliers), write a cleaning script + visualization script, run and
verify, summarize your approach"
# measured: 52→35 rows zeroed + chart + trade-off notes (186s)
```

Highlight: dsh doesn't just execute — it explains trade-offs ("median imputation creates a spike", "whether to drop outliers depends on business") — direct evidence of an agent that *thinks*.

5.4 Scenario 3: Documentation & knowledge work

```
dsh --profile headless "Generate complete Markdown API docs for user_module.py (8 functions):
params, returns, exceptions, examples, edge cases"
```

```
# measured: 423-line ReadTheDocs-style doc (1m04s), proactively flags contract-vs-
implementation gaps
```

Highlight: defensive writing — notes calling traps (negative limit behavior, empty-argument behavior), flags where docs and code disagree.

5.5 Scenario 4: Automation & ops (headless is the ace)

```
# cron daily digest
dsh --profile headless "Read today's git log in the workspace and write a Chinese daily
summary" > daily-report.md

# batch: convert all .txt in docs/ to .md
dsh --profile headless "Convert all .txt files under docs/ to .md (preserve structure)"
```

Why headless is the ace: full process-level tools + non-zero exit on failure + pipeable output — **an agent you can put in CI**.

5.6 Scenario 5: Code understanding & refactoring

```
dsh --profile headless "Refactor legacy_orders.py to OOP with separated responsibilities,
keep behavior identical, write tests verifying output parity"
# measured: 17/17 tests pass incl. script-level artifact comparison (4m37s)
```

Highlight: engineering awareness — merges duplicated logic into one entry point, keeps backward compatibility, **notices a sandbox tempfile-permission issue and switches approach**.

5.7 Three prompt rules that make scenarios work

1. **State acceptance criteria, not process**: "run and verify" > "write a script" — dsh figures out the steps.
2. **Give context**: one sentence of background (where files are, what the data looks like, who the audience is).
3. **One task at a time**: split big goals into rounds of small closed loops.

5.8 Your first real task (15 minutes)

1. Make a test dir with a small file (code / data / notes).
2. Run: `dsh --profile headless "Summarize this file and list 3 improvement points"`.

3. Run: `dsh --profile headless "Write a test / generate docs for this file"`.
4. Compare output — did it give a real artifact, or a vague answer? Were your acceptance criteria clear enough?

5.9 Industry view: how dsh opens in different domains

Same toolchain, different industries. Generic scenarios only (no real business data); regulated fields (medical/legal) require human review.

Finance & research

`dsh --profile headless "Read this financial data and generate a summary with YoY/MoM changes"` — data pipeline + high cache hit make long-session reports cheap.

Education & learning

`dsh --profile headless "Turn this chapter into 10 self-test questions with explanations"` — document generation + structuring are strong; batch question generation.

Sales & support

`dsh --profile headless "Group 100 customer feedback entries by issue type and suggest improvements"` — batch text processing + classification.

Operations & content

`dsh --profile headless "Read this week's data files and generate an ops weekly report (with chart code)"` — data + docs + automation combined.

Research & review

`dsh --profile headless "Compare the methods and conclusions of these 3 papers, output a comparison table"` — long-doc processing + structured output (1M context).

⚠ Compliance note

High-risk domains (medical diagnosis, legal advice, financial advice): dsh output needs human review — tools are sandboxed, but **content responsibility is on the user**.

Hands-on exercises

1. **Scenario mapping**: classify 3 of your daily tasks into the five scenarios.
2. **Prompt comparison**: write two prompt versions for the same task — one describing process, one stating acceptance criteria — run both and compare.
3. **Headless scripting**: turn "daily git digest" into one cron/bash command; verify output.

4. **Multi-step task**: design a cross-scenario task (read code → generate docs → output report) and watch dsh orchestrate.

5. **Think**: which scenarios is dsh NOT suitable for? (Hint: real-time interaction, sensitive operations needing human approval.)

FAQ

- **Q: Is dsh just a coding assistant?** No — its toolchain (read/write/run/search) makes it general; data/docs/automation all verified.
- **Q: How long do complex tasks take?** 1–5 min measured (V4-Flash); lower reasoning effort for speed (Ch.6).
- **Q: What if a task fails?** Trace the tool calls in output; usually the acceptance criteria were unclear — rewrite and rerun.
- **Q: Can it run automated?** Yes — headless + non-zero exit + pipeable output works in cron/CI.
- **Q: Is it safe?** Tool execution has sandbox + approval layers; sensitive ops should be human-confirmed (Ch.8).
- **Q: How is this different from using Claude Code directly?** Same-model capability is close, but dsh lets you customize toolchains/UI/plugins (Ch.1 matrix).

Chapter 6: Advanced & Performance Tuning

> ***Goal of this chapter:*** *Go from "it runs" to "it runs well" — reasoning effort strategy, tool-call latency analysis, and a real-world pitfall checklist.*

TL;DR (30-second version)

1. **Where the time goes**: model thinking takes ~90%, tool execution <1%, network/rendering ~10%. Optimizing thinking time is the highest-leverage speedup.
2. **Three-tier strategy**: `low` (simple/batch turns), `high` (everyday default), `max` (complex reasoning/debug). Choose manually via the UI or let a plugin adjust automatically.
3. **Latency visualization**: the stats line at the bottom of the Web UI (`LLM Xs / Tool calls Ys`) is the fastest way to locate bottlenecks.
4. **Seven real pitfalls**: rc.1 dependency break, missing plugin `main`, un-awaited `next()`, unrecognized event types, client tests that won't run, cache-hit misattribution, port conflicts.
5. **Evaluation three questions**: who tested it, which harness, how strict is the verifier. Always read benchmarks with context.

6.1 Performance Model: Where Does dsh Spend Its Time

Measured latency breakdown for a "create a file" task:

Phase	Share	Notes
Model thinking (Think)	~90%	**Re-thinks before every tool call** — the overwhelming majority
Tool execution	<1%	File writes and similar, millisecond-level
Network / rendering	~10%	API round-trip + UI updates

Implications:

- Simple tasks → optimize thinking time (lower reasoning effort)
- Long tool-chain tasks → save thinking time at every step; cumulative gains are largest
- Slow tools themselves (search / large files) → optimize the tool implementation, not the effort level

6.2 reasoning_effort Strategy (Official Three Tiers)

Tier	Recommended Use
`low`	Simple / deterministic turns: file operations, batch jobs, cheap steps in a tool chain
`high`	Everyday agent tasks (default)
`max`	Complex reasoning, long-chain planning, debugging

Manual: The "Reasoning Level" selector in the UI, or `reasoningEffort` in `~/.dsh/settings.yaml`.

Automatic: A plugin dynamically downgrades per tool turn (example speed-up plugin, Chapter 4).

6.3 Tool-Call Latency Visualization

dsh's session stats line (at the bottom of the Web UI) shows: `N turns · M steps | LLM Xs · Tool calls Ys | Avg first token ...` — this is the fastest way to locate bottlenecks.

Advanced: A host plugin can listen to tool events for per-tool timing (the example speed-up plugin already includes a host-side logging version), surfacing "which tool is the slowest."

6.4 Pitfall Checklist (Battle-Tested, with Fixes)

#	Pitfall	Symptom	Fix	
1	rc.1 broken dependency chain	`pnpm install` 404 (`dsh-type-meta` etc. were never published)	Use the `^0.1.0-rc.6` dependency line	
2	Plugin missing `main`	`No "exports" main defined`	Expose a `.` entry; `"main": "src/index.ts"` can be loaded by tsx	
3	`next()` not awaited	Provider/model lost, error thrown	`next()` in `agent/request` returns a Promise; must await	
4	Event type not recognized	`'agent/request' is not assignable to keyof Events`	npm doesn't re-export type augmentations; relax the signature at the boundary	
5	Client tests won't run	jsdom reports `window.__ModuleLoader__` undefined	Client artifacts depend on dsh's bootstrap mechanism; run component tests in official CI	
6	Misattributing "suddenly faster" on simple tasks	1s vs 110s difference misattributed	DeepSeek context cache hits also speed things up — A/B tests must use a fresh prompt	
7	Port conflict	`dsh web` won't start	`netstat -ano`	findstr 3080` to find the PID, then kill it

6.5 Evaluation Perspective: Official Scorecards vs. Independent Benchmarks

(With the 0813 official release in mind) When reading agent model scorecards, ask three questions:

1. **Who tested it?** Official self-tests (their own harness) vs. independent third parties (AA, etc.)
2. **Which harness?** Different frameworks yield very different scores (official Terminal-Bench 87.9 vs. AA independent 79)
3. **How strict is the verifier?** Lenient verifier (SWE-bench Verified has 8.5% false positives) vs. strict (DeepSWE 0.3%)

dsh's `agent/request` waterfall makes model benchmarks reproducible. That's an engineering advantage over closed-source products.

Hands-on exercises

1. **Measure your own task**: run a simple file-creation task in `dsh web`. Check the stats line at the bottom. What's the LLM time vs tool-call time? Does it match the 90% / <1% split?
2. **Effort comparison**: run the same task twice, once with `reasoningEffort: low` and once with `high`. Compare wall-clock time and output quality. When is `low` good enough?
3. **Pitfall hunt**: open the pitfall checklist (Section 6.4). For each of the 7 pitfalls, try to reproduce it (or find a log/example where it occurred). Write down the fix.
4. **Plugin timing**: if you've built the example speed-up plugin (Chapter 4), check its logs. How much time does each step save? Calculate the cumulative gain over a 50-step task.
5. **Benchmark skepticism**: find an official DeepSeek benchmark scorecard. Ask the three evaluation questions (Section 6.5). What harness did they use? How strict was the verifier?
6. **Think**: why does dsh re-think before every tool call? What would happen if it cached the reasoning across steps? What are the trade-offs?

FAQ

- **Q: Why is model thinking 90% of the time?** Because dsh uses a reasoning model that generates a chain-of-thought before every action. In a multi-step task, each step triggers a new reasoning pass. This is by design for quality, but it's slow.
- **Q: Can I just set `reasoningEffort: low` forever?** For simple, deterministic tasks (file ops, batch jobs), yes. For complex reasoning, planning, or debugging, you'll get better results with `high` or `max`. The plugin approach (Chapter 4) gives you dynamic adjustment.
- **Q: Why did my task suddenly run 100x faster?** Likely a context-cache hit. DeepSeek's API caches repeated input tokens at a 98% discount. If your prompt/history is similar to a previous request, most tokens hit cache. This is a good thing, but it makes A/B testing tricky (use fresh prompts).

- **Q: What's the "overflow agent" in compaction?** When a conversation exceeds the context window and compression fails, dsh routes to a fallback "overflow agent" that tries to salvage the task. It's a last-resort mechanism.
- **Q: How do I read the stats line?** `N turns · M steps | LLM Xs · Tool calls Ys | Avg first token ...` — LLM time is total model thinking, tool-call time is total tool execution. If LLM time dominates, lower the effort. If tool time dominates, optimize the tool.
- **Q: Are official benchmarks reliable?** They're a starting point, but always ask: who tested it, which harness, how strict is the verifier. Independent benchmarks with strict verifiers are more trustworthy.

Next chapter: Chapter 7: Ecosystem & Resources (planned).

Chapter 7: Ecosystem & Resources

> ***Goal of this chapter:*** *Give you a map for "joining the dsh ecosystem" — official entry points, the state of the community, and how to participate.*

TL;DR (30-second version)

1. **Official entry points**: GitHub repo (source + issues), API docs, Discord, Discussions. The current contribution entry is primarily via Discussions.
2. **External PRs are not accepted yet** (as of 2026-08-13), but the official team encourages dsh-plugin ecosystem projects, tutorials, and blog posts.
3. **Plugin ecosystem snapshot**: `DSH-better-sidebar` (most complete community plugin), an example speed-up plugin (Chapter 4 full breakdown), this handbook. Search `topic:dsh-plugin` to discover more.
4. **Beginner participation path**: use it → make small improvements (PRs to community plugins) → publish your own plugin → write content.
5. **Ecosystem day-zero = first-mover advantage**: every early ecosystem has a "first crab" bonus. Now is the time to get in.

7.1 Official Entry Points

Resource	URL	Purpose
Official repository	https://github.com/deepseek-ai/deepseek-harness	Source code, architecture docs, issues
API documentation	https://api-docs.deepseek.com	Models, pricing, API guides
Discord	Link in official README	Community discussion
Discussions	Official repo Discussions	Proposals / help requests (currently the recommended contribution entry point)

Note: The official CONTRIBUTING guide states "external PRs are not accepted at this time" (as of 2026-08-13). However, the following are encouraged:

- Submit suggestions in Discussions (the team evaluates them)
- **Build dsh-plugin ecosystem projects** (an officially endorsed contribution path)
- Write tutorials and blog posts

7.2 Plugin Ecosystem (Snapshot as of 2026-08-13)

Project	Focus	Status
`DSH-better-sidebar`	File management / terminal / Git / browser sidebar	Most complete community plugin

Example speed-up plugin	Tool-call speed-up (automatic reasoning_effort adjustment)	Teaching example (Chapter 4)
`dsh-handbook` (this handbook)	Beginner tutorial	Ecosystem documentation

Discovering plugins: Search GitHub for `topic:dsh-plugin`.

Publishing a plugin: Add the `dsh-plugin` topic to your repo + publish on npm.

7.3 How to Join the Ecosystem (Beginner Path)

1. **Use it:** `dsh web` + install two community plugins, integrate into your daily workflow
2. **Small improvements:** Submit PRs to community plugins (read the three cases in Chapter 5 for a complete PR paradigm)
3. **Publish a plugin:** Start from the minimal host plugin in Chapter 4, tag it with `dsh-plugin`
4. **Write content:** Tutorials / reviews / pitfall guides (officially encouraged); cross-reference this handbook

7.4 Recommended Reading Path

Goal	Path
Quick start	Chapter 2 → install better-sidebar → use daily
Plugin development	Chapter 3 → Chapter 4 → follow the Chapter 5 cases
Performance tuning	Chapter 6 → Chapter 4 example source
Deep customization	Official AGENTS.md (architecture) → docs/architecture.md → packages/ source code

Closing Words

dsh was open-sourced on 2026-08-13. **Every day of the ecosystem is "early days."** This handbook will keep updating as dsh evolves. If a command in some chapter stops working, it's most likely due to rc version iteration. Always defer to the official changelog.

Hands-on exercises

1. **Explore the official repo**: go to https://github.com/deepseek-ai/deepseek-harness. Read the README, then open `packages/AGENTS.md`. What architecture insights can you find?
2. **Join the community**: join the Discord or browse the Discussions tab. What topics are people talking about? What questions are unanswered?
3. **Install a community plugin**: if you haven't already, install `dsh-better-sidebar` or build the example speed-up plugin from Chapter 4. Use it for a day. What works well? What could be improved?

4. **Submit a suggestion**: go to the official Discussions tab. Write a proposal for a feature or improvement. Be specific: what problem does it solve? How would it work?

5. **Publish a plugin**: start from the minimal host plugin in Chapter 4. Add a feature (e.g. a new tool, a UI tweak). Tag it with `dsh-plugin` and publish to npm.

6. **Think**: why does the official team encourage ecosystem projects instead of accepting PRs? What are the advantages and risks of this approach?

FAQ

- **Q: Can I contribute code to the official repo?** Not via PR at the moment (2026-08-13). The official team encourages ecosystem projects (plugins, tutorials, tools) instead. Check the CONTRIBUTING guide for updates.
- **Q: How do I discover community plugins?** Search GitHub for `topic:dsh-plugin`. The ecosystem is growing fast, so new plugins appear regularly.
- **Q: What's the best way to get help?** Join the Discord or post in the official Discussions tab. The community is active and responsive.
- **Q: Do I need to publish my plugin to npm?** No, but it's recommended. If you want others to use it easily, publish to npm and tag it with `dsh-plugin`.
- **Q: Is it too late to join the ecosystem?** No. The ecosystem was open-sourced on 2026-08-13. Every day is "early days." First-movers have an advantage, but the ecosystem is still small enough that quality contributions stand out.

Go claim your spot in this brand-new ecosystem. 🚀

Chapter 8: Tools & Context System

> ***Goal of this chapter:*** *Understand dsh's "capability engine" — which tools the model can call, how context is fed to the model, and how long conversations are handled.* ***This is the key chapter for going from "it runs" to "I understand how it runs."***

TL;DR (30-second version)

1. **60+ official capability packages**: tools (fs/shell/web/skill/todo), context (context/compaction), session, subagent, MCP, workflow, safety, model, UI. Everything is a plugin.
2. **Built-in tool names are short verbs**: `read`/`write`/`grep`/`glob`/`edit`/`bash`/`todo`/`skill`. When writing prompts, just say "read the file" or "search" and it works.
3. **Tool returns carry `locations` → artifact tracking**: the model can see which files it changed, and the UI lets you open them directly (artifact chips at the end of the conversation).
4. **Context = system prompt + skill catalog + conversation history + tool results**: layered injection, with tool schemas carried in every request.
5. **Long conversations are auto-compressed (compaction)**: detect overflow → prune history → optional summarization → fallback to overflow agent. Important info should be written into the prompt manually.

8.1 Official Capability Map (60+ Packages at a Glance)

All of dsh's capabilities are provided as packages (`packages/<group>/<name>`). The ones newcomers need to know first:

Capability Domain	Official Packages	Purpose
Tools	`fs/tool-fs`, `fs/tool-fs-search`, `fs/tool-str-replace-editor`, `shell/tool-bash`, `web`, `skill`, `todo`	Callable tools for files, terminal, web, skills, todos, etc.
Context	`context/*`, `compaction/*`	Request context assembly, long-conversation compression
Session	`session/*`	Persistence, titles, telemetry
Subagent	`subagent/*`	Delegate sub-tasks
MCP	`mcp/*`	MCP client (external tool servers)
Workflow	`workflow/*`	Multi-step workflow orchestration
Safety	`sandbox/*`, `guard/*`, `interaction/*`	Sandboxing, loop hygiene, permissions/approvals
Model (LLM)	`llm/*`, `llm-deepseek`, `llm-retry`	Model integration, retries
Skill	`skill/*`	Skill provider registry
Client (UI)	`client/*` (ui-conversation, ui-tool, …)	Web UI components

Full list: see `packages/AGENTS.md` in the official repository.

8.2 Built-In Tools (Observed in Practice)

The actual tool names the model can call are **short verbs** (observed from dsh web sessions and agent/request logs):

Tool Name	Purpose	Notes
`read`	Read files	fs capability
`write`	Write files	fs capability
`grep`	Content search	fs-search
`glob`	File pattern matching	fs-search
`edit` / `str_replace_editor`	Precise editing	Tool results include `locations` (used for artifact tracking)
`bash` / `pwsh`	Execute commands	Sandbox isolation
`todo`	Todo management	Long-task planning
`skill`	Skill invocation	Injected via skill-catalog

Tool results and artifact tracking (important concept): Tool return values carry `locations` (file paths). dsh uses these to build "artifact file lines" — the artifact chips at the end of a conversation come from here. **The model can see which files were changed, and the UI lets you open them directly.**

8.3 How Context Is Fed to the Model

A single model request's context = system prompt + skill catalog + conversation history + tool results. This is visible in session logs:

```
Context injection @deepseek-ai/dsh-system-prompt   ← Official system prompt
Context injection skill-catalog                    ← Skill catalog
```

dsh's context mechanism:

- **Layered system prompts:** Official plugins register prompt sections via `systemPrompt.section()` (e.g. ui-deliverables registers "artifact file reference" guidance)
- **Skill catalog injection:** The list of available skills enters the context; the model calls them as needed
- **Tool schemas:** Every request carries tool definitions

8.4 Long Conversations: Compaction

Long conversations blow up the context window. dsh's `compaction` plugins (e.g. `compaction-basic`) handle:

- Detecting context overflow (`CONTEXT_WINDOW_EXCEEDED` via `agent/request-error`)

- Compressing history (model-agnostic pruning + optional summarization)
- Routing to an "overflow agent" on failure

*For newcomers: **Just know that "long conversations are auto-compressed."** The details are an advanced topic. In production, note that compression loses detail — important context should be written into the prompt manually.*

8.5 Permissions & Security Model (Awareness Level)

- **Access modes:** The UI shows modes like "Workspace Write" (permission presets)
- **Interactive approvals:** `interaction/*` provides permission/approval capabilities (dangerous operations can require confirmation)
- **Sandboxing:** `sandbox/*` isolates command execution (e.g. the pwsh sandbox has ACL constraints — temp directory permission issues have been observed in practice)
- **Tool timeouts:** `guard/*` provides loop hygiene and tool timeouts

*Deep security configuration is beyond the scope of this handbook. The key takeaway: **dsh's tool execution has isolation and approval layers by default.** It's not bare execution.*

8.6 Three Things Newcomers Should Remember Most

1. **Tool names are short verbs** (read/write/grep/glob/edit/bash) — when writing prompts or plugins, just say "read the file" or "search" and it works
2. **Tool returns include `locations` → artifact tracking** — files the model changed appear in the conversation's artifact area
3. **Long conversations are auto-compressed** — no need to manually clear history (but important info should go into the prompt)

Hands-on exercises

1. **Tool inventory**: open a `dsh web` session. Ask the model: "List all the tools you can call." Compare the list with Section 8.2. Are there any surprises?
2. **Artifact tracking**: give dsh a task that modifies multiple files (e.g. "Create a Python project with a main script, a test file, and a README"). After the task, check the artifact chips at the end of the conversation. Can you open each file?
3. **Context inspection**: open the session log (top-right in `dsh web`). Look for "Context injection" lines. What sections are injected? How much of the context is system prompt vs conversation history?
4. **Long conversation test**: have a 20+ turn conversation with dsh. At what point does compaction kick in? Check the logs for compaction events. Does the model still remember early context?

5. **Permission modes**: in the Web UI, switch between different access modes (e.g. "Workspace Write"). What changes? What operations are restricted?

6. **Think**: why are tool names short verbs instead of descriptive names? How does this affect the model's ability to use them correctly?

FAQ

- **Q: What's the difference between `edit` and `write`?** `write` creates or overwrites a file. `edit` (or `str_replace_editor`) makes precise replacements within a file. Use `edit` for small changes, `write` for new files or major rewrites.
- **Q: Why does the model sometimes use `bash` instead of `read`?** The model chooses tools based on the task. If it needs to run a command (e.g. `cat file.txt`), it uses `bash`. If it just needs to read the file content, it uses `read`. Both work, but `read` is more efficient.
- **Q: What happens when the context window fills up?** dsh's compaction plugins detect the overflow, prune the history, and optionally summarize. You don't need to manually clear the conversation, but important context may be lost. For critical info, write it into the prompt.
- **Q: Can I add custom tools?** Yes, via a host plugin. Use the `tools` capability to register new tools. The model will see them in the tool schema and can call them.
- **Q: What's a "skill" and how is it different from a tool?** A skill is a higher-level capability injected via the skill catalog. The model calls skills via the `skill` tool. Skills are typically more complex than tools (e.g. "review this code" vs "read this file").
- **Q: Is sandboxing enabled by default?** Yes. Tool execution has isolation and approval layers. Dangerous operations (e.g. `bash` commands) may require confirmation. You can configure the sandbox via the `sandbox/*` and `interaction/*` packages.

Next chapter: Chapter 9: MCP, Subagents & Workflows

Chapter 9: MCP, Subagents & Workflows

> ***Goal of this chapter:*** *Understand dsh's three extension capabilities — MCP (connecting external tools), subagents (parallel tasks), and workflows (multi-step orchestration).* ***These are the switches that upgrade dsh from a "single agent" to an "agent system."***
>
> ⚠ *This chapter is based on official architecture docs (`packages/AGENTS.md`) and package structure. Some feature examples are marked "pending live testing" — rc versions iterate quickly, so defer to the official changelog for exact usage.*

TL;DR (30-second version)

1. **MCP = plug external tools into the agent**: connect to databases, browsers, internal systems via the MCP protocol. Get the built-in tools working first, then add MCP when you have a clear gap.
2. **Subagents = work in parallel**: delegate tasks to sub-agents for parallel execution (large-repo research, long-task decomposition, independent verification). Master single-agent workflow first.
3. **Workflows = deterministic process orchestration**: unlike "multi-turn conversation" (model freestyles), workflows define steps as a flow executed in order or by condition. Suited for fetch → clean → report → validate.
4. **Combined = agent system**: parent agent plans → subagents research in parallel → tool chain + MCP → workflow aggregates → artifacts.
5. **Beginner path, four stages**: single agent + built-in tools → + MCP → + subagents → + workflows. Stack gradually, don't skip levels.

9.1 MCP: Connecting to the External Tool Ecosystem

MCP (Model Context Protocol) is an open protocol for "plugging external tools into an agent." dsh provides an `mcp` client package (`@deepseek-ai/dsh-mcp-client`).

What it does: Connect to any MCP server via MCP (databases, browsers, charts, internal company systems, etc.). For example, the community's `dsh-plugin-cost-tracker` (token tracking) is an MCP/plugin hybrid.

Positioning for newcomers: MCP is an "ecosystem extension." Only reach for it once you have a clear tool gap inside dsh. Get the built-in tools working first, then add MCP.

9.2 Subagents: Working in Parallel

What they are: Delegate tasks to sub-agents for parallel execution (`packages/subagent/*`).

Typical scenarios:

- Multi-module research in a large repo (one subagent per module)
- Long-task decomposition (parent agent plans, subagents execute)
- Independent verification (subagents cross-check each other)

For newcomers: Subagents are "advanced weaponry." Master the single-agent workflow first, understand task decomposition, then move to subagents. A common anti-pattern is "spawning a subagent for everything," which just adds coordination overhead.

9.3 Workflows: Multi-Step Orchestration

What they are: `packages/workflow/*` provides multi-step workflow orchestration (worker-thread provider + tool Consumer).

How they differ from "multi-turn conversation": A normal conversation is "the model freestyles." A workflow is "steps are defined as a flow, executed in order or by condition." This suits **deterministic processes** (e.g. fetch data → clean → generate report → validate).

Official examples (`packages/examples/`): Runnable cordis.yml workflow samples are available.

9.4 Putting It Together: What an "Agent System" Looks Like

```
Your prompt (goal)
  ↓
Parent Agent (planning)
  ├── Subagent A: Research module A (parallel)
  ├── Subagent B: Research module B (parallel)
  └── Tool chain: read/grep/bash + MCP (database)
  ↓
Workflow (e.g. validate → aggregate → output report)
  ↓
Artifacts (openable / trackable)
```

Suggested beginner path:

1. **Stage 1:** Single agent + built-in tools (Chapters 2–5)
2. **Stage 2:** + MCP (connect external tools)
3. **Stage 3:** + Subagents (parallelism)
4. **Stage 4:** + Workflows (deterministic flows)

9.5 Community Ecosystem Examples (Snapshot as of 2026-08-13)

Community projects already appearing in the official Discussions:

- `dsh-plugin-cost-tracker` — real-time token cost tracking (plugin/MCP hybrid)

- Plugin development / speed-up tools (e.g. the example speed-up plugin in Chapter 4 of this handbook)

The ecosystem is growing fast. **Now is a great time to start building.**

Hands-on exercises

1. **MCP exploration**: search GitHub for MCP servers (e.g. database, browser, chart). Pick one. Read its documentation. What would it add to dsh?
2. **Subagent design**: think of a task that could benefit from parallelism (e.g. "research 5 modules in a large repo"). How would you decompose it into subagent tasks? What would the parent agent do?
3. **Workflow sketch**: design a deterministic workflow for a real task (e.g. "daily data pipeline: fetch → clean → analyze → report"). What are the steps? What conditions or branches exist?
4. **Agent system diagram**: draw a diagram of an "agent system" that combines all four capabilities (single agent + MCP + subagents + workflow). What does each component do?
5. **Beginner path check**: which stage are you at? If you're at stage 1 (single agent), master it first. If you're at stage 2 (MCP), what external tool are you connecting? What gap does it fill?
6. **Think**: why does the guide say "don't skip levels"? What goes wrong if you spawn subagents before mastering single-agent workflow?

FAQ

- **Q: When should I add MCP?** Only when you have a clear tool gap. If dsh's built-in tools (read/write/bash/grep) can't do what you need (e.g. query a database, interact with a browser), then MCP is the answer.
- **Q: Do subagents speed up every task?** No. Subagents add coordination overhead. They're useful for parallelizable tasks (research, decomposition, verification). For sequential tasks, they just add complexity.
- **Q: What's the difference between a workflow and a multi-turn conversation?** A conversation is "the model freestyles." A workflow is "steps are defined as a flow." Workflows are deterministic; conversations are adaptive. Use workflows for repeatable processes.
- **Q: Can I combine subagents and workflows?** Yes. A workflow can spawn subagents for parallel steps. This is the "agent system" pattern: parent plans, subagents execute, workflow orchestrates.
- **Q: Are subagents and workflows stable in rc?** They're part of the official architecture, but rc versions iterate quickly. Check the changelog and `packages/AGENTS.md` for the latest status.
- **Q: Where do I find MCP servers?** Search GitHub for "MCP server" or check the official MCP documentation. The ecosystem is growing fast.

Appendix A: Glossary & Command Quick Reference

Chapter 10: Complex Real-World Cases (Run Live in dsh)

> ***Goal of this chapter:*** *Demonstrate dsh's actual capabilities, output quality, and timing on multi-step tool chains through two* ***complex tasks completed live in dsh.***
> ***Privacy notice:*** *Both cases use* ***synthetic data / self-authored code*** *only. No real business data or sensitive information is involved.*

TL;DR (30-second version)

1. **Case A (data quality analysis, 186 seconds)**: 52 rows of dirty data → analysis → cleaning script → visualization → run verification. 52 → 35 rows, all issues resolved.
2. **Case B (5-bug fix + 49 tests, 94 seconds)**: calculator module with 5 planted bugs → all fixed → 49 tests pass, covering edge cases, precision, and exceptions.
3. **dsh's profile**: auto-orchestrated multi-step tool chains, shows judgment (proactively explains data trade-offs), artifact tracking, controllable timing (94-186s).
4. **Key insight**: write acceptance criteria clearly in the prompt (e.g. "run and verify"), and dsh will close the loop. For complex tasks, specify "what to do + how to verify."
5. **Failure recovery**: not triggered in these cases. Production recommendation: pair with guard timeouts + human approval.

Case A: Data Quality Analysis + Cleaning + Visualization (186 seconds)

Task

Give dsh a synthetic JSON file with dirty data (52 rows: missing values, type errors, duplicate rows, outlier value 99999), and ask it to:

1. Analyze data quality issues
2. Write a cleaning script `clean.py`
3. Write a visualization script `visualize.py` (generating `chart.png`)
4. Run both scripts to verify

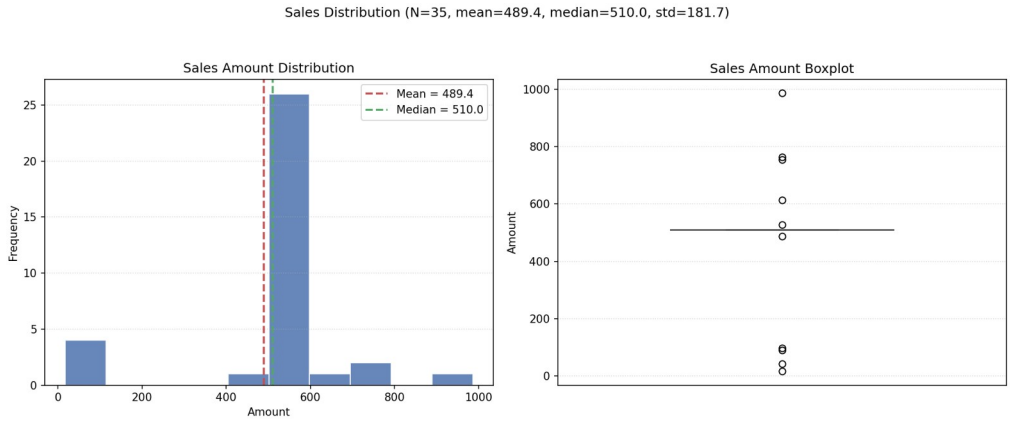
How dsh Did It (Tool Chain)

```
read(sales_data.json) → write(clean.py) → bash(python clean.py)
→ write(visualize.py) → bash(python visualize.py) → read(output) → summary
```

Output & Verification

Artifact	Result
`clean.py`	✅ Runs successfully, 52 → **35 rows**, all issues resolved

`visualize.py` + `chart.png`	✅ Valid PNG generated (1950×825)
Cleaning strategy	Median imputation for missing amounts, outlier removal, deduplication, type normalization



Output Highlights (Shows Judgment, Not Just Mechanical Execution)

- Median imputation creates a 510 spike in the histogram — an inherent side effect of imputation, worth noting

- If 99999 represents a genuine large order, consider IQR/quantile methods instead of blanket deletion

- Post-cleaning stats: N=35, mean 489.43, median 510, std dev 181.73

dsh didn't just execute the task — it proactively explained the trade-offs and assumptions in data processing. This is evidence of "thinking" at the agent engineering layer.

Case B: 5-Bug Code Fix + 49 Tests (94 seconds)

Task

Give dsh a Python calculator module **deliberately seeded with 5 bugs**, and ask it to: find all bugs → fix them → write comprehensive unit tests → run and verify.

The Planted Bugs

1. add mistakenly written as `a - b`
2. `divide` silently returns 0 on division by zero (should throw)
3. `factorial` silently returns -1 for negative input (should throw)
4. `factorial`'s `range(1, n)` misses multiplying by n
5. `format_money` doesn't format to two decimal places

dsh's Fixes + Tests (Verification: 49 passed)

Full test file: case-test-calculator.py — 49 test cases covering:

Function	Coverage

`add`	Positive / negative / zero / float / precision / commutativity
`subtract`	Negative results / zero boundary / float
`multiply`	Sign combinations / multiply by zero / float
`divide`	Integer division / signs / float / **division by zero must throw ZeroDivisionError**
`factorial`	0!/1! boundary / known values / **negative must throw ValueError**
`format_money`	Zero-padding / rounding / **0.1+0.2 rounding** / negative

```
python -m pytest test_calculator.py -v   # 49 passed in 0.37s
```

Observations: dsh's Performance

- **Full closed loop:** Read → fix → write tests → run tests → summarize, with no human intervention
- **Fix quality:** All 5 bugs fixed, with docstring explanations added
- **Test design:** Covers edge cases (division by zero / negatives / precision), not just "does it run"

Case Summary: dsh's Profile on Complex Tasks

Dimension	Performance
Multi-step tool chain	✅ Auto-orchestrated (read → write → run → verify → summarize)
Complex task timing	94s–186s (same model, same gateway; slower than benchmark simple tasks but manageable)
Output quality	✅ Shows judgment (data trade-off explanations, test edge-case design)
Artifact tracking	✅ Generated files can be opened directly at the end of the conversation
Failure recovery	Not triggered in these cases; production recommendation: pair with guard timeouts + human approval

Takeaway for newcomers: dsh excels at tasks involving "multiple files, multiple steps, and verification" — which is also its primary value proposition as an agent runtime. For complex tasks, the recommendation is to **write acceptance criteria clearly in the prompt** (e.g. "run to verify"), and dsh will follow through to completion.

Hands-on exercises

1. **Reproduce Case A**: create a synthetic JSON file with dirty data (missing values, type errors, duplicates, outliers). Give it to dsh with the same prompt. Compare your results with the case study.
2. **Reproduce Case B**: create a Python module with 5 planted bugs. Ask dsh to find and fix them, then write tests. How many bugs does it find? How many tests does it write?

3. **Prompt variation**: run Case A twice. First time, say "clean the data." Second time, say "clean the data, run and verify, and explain your trade-offs." Compare the output quality.

4. **Timing analysis**: measure how long each step takes (read, write, bash, verify). Which step is the bottleneck? How does this compare to the 90% / <1% split from Chapter 6?

5. **Failure injection**: give dsh an impossible task (e.g. "fix this code, but don't run the tests"). Does it still try to verify? What happens when the acceptance criteria are unclear?

6. **Think**: why does dsh "show judgment" (explain trade-offs, design edge-case tests)? Is this a feature of the model, the harness, or the prompt?

FAQ

- **Q: Are these real-world cases or synthetic?** Synthetic. The data and code are self-authored to demonstrate capabilities. No real business data is involved.
- **Q: Why 94 seconds for Case B?** That's the total wall-clock time for reading the code, finding 5 bugs, fixing them, writing 49 tests, running them, and summarizing. Most of the time is model thinking (Chapter 6).
- **Q: What if dsh misses a bug?** In Case B, it found all 5. But in general, the quality depends on the prompt. If you say "find all bugs," it will try. If you say "fix the obvious bugs," it might miss subtle ones.
- **Q: Can I use dsh for production data cleaning?** Yes, but review the output. dsh explains its trade-offs (e.g. "median imputation creates a spike"), but you should verify the cleaning logic matches your business rules.
- **Q: What's the "artifact tracking" mentioned in the summary?** Tool returns carry `locations` (file paths). dsh uses these to build "artifact file lines" — the artifact chips at the end of the conversation. You can click to open each file.
- **Q: How do I handle failure recovery in production?** Pair dsh with guard timeouts (to prevent infinite loops) and human approval (for sensitive operations). The `guard/*` and `interaction/*` packages provide these capabilities.

Appendix A: Glossary & Command Quick Reference

Chapter 11: Future Outlook — Where dsh and the Agent Ecosystem Are Headed

*Goal: project the future of dsh and its ecosystem from multiple angles — **technology, ecosystem, competition, industry, developer opportunity, risk**. These are projections based on current facts, not assertions — use them to judge whether it's worth investing.*

TL;DR (30-second version)

1. **Short term (1–3 months)**: official 0.1.0 release, plugin ecosystem boom, tutorial/tool projects grab the first wave of dividends
2. **Mid term (1 year)**: dsh becomes a candidate de-facto standard for the "programmable agent base", vertical plugins mature
3. **Long term (2–3 years)**: agent engineering layer standardizes; dsh ecosystem and model iteration reinforce each other
4. **Biggest risks**: breaking changes, ecosystem fragmentation, official strategy shifts
5. **Individual opportunity**: entering now (tutorials/plugins/tools) is cheap early-mover dividend

<details><summary>Chapter navigation</summary>

11.1 Technology projections

Horizon	Projection	Basis
Short	`0.1.0` stable release (drop rc), breaking changes converge	Currently rc.6; official explicitly warns of breaking changes; iteration is fast
Short	Official TUI & interactive CLI (highest community demand, #172 15+ comments)	Hottest request in official Discussions

Mid	Cache/performance optimization becomes the theme (finer reasoning_effort, cache-hit tooling)	High cache-hit rate is already a cost advantage (ch. 5); it will be made explicit via tooling
Mid	Plugin ecosystem standardization (marketplace / one-click install / quality tiers)	Official `dsh-plugin` topic exists; plugin growth inevitably creates a marketplace
Long	Agent runtime architecture solidifies as an "engineering paradigm" (an Agent-era analog of the Linux kernel model)	"Everything is a plugin" has proven to be a composable architecture

11.2 Ecosystem projections

- **Plugin count**: day zero → tens to hundreds within a year (referencing similar ecosystem S-curves)
- **Content ecosystem**: tutorials/blogs/awesome lists erupt first (Chinese content is currently a vacuum — this handbook is the first systematic tutorial)
- **Community size**: official Discussions adds dozens of threads daily (we observed 30+ new topics in a single day); Discord activity will grow in tandem
- **Flagship projects**: a few "ecosystem flagships" will emerge (think Prettier/ESLint status in the VSCode ecosystem) — **tool-type and tutorial-type are most likely**

11.3 Competitive landscape

Dimension	How dsh's positioning evolves
vs Claude Code	Short term still "less out-of-the-box"; long term differentiates on customization + open source + cost
vs OpenCode	Both open source; dsh's official backend + plugin ecosystem is the differentiator (OpenCode has no official backend)
vs building your own	dsh is the bridge from "half-finished to finished" — less work than DIY, more control than closed source
Model side	Each DeepSeek model generation amplifies dsh's ecosystem value (model + runtime synergy)

Key judgment: dsh's winning move isn't "how strong the model is" (that's DeepSeek's job) — it's whether the **agent engineering layer can become the standard**. Whoever defines "how plugins are written, how the ecosystem grows" wins that layer.

11.4 Industry applications

- **Finance/data analysis**: long sessions + high cache-hit = cost-friendly batch analysis, first to land
- **Enterprise knowledge/docs**: 1M context + document toolchain makes internal knowledge management a natural fit

- **Education**: batch problem generation/explanation/summarization — cost-sensitive scenarios eat the cache dividend
- **Vertical agent products**: "industry agent wrappers" will emerge (industry plugins/config bundles built on dsh)

11.5 Developer opportunities (for those who enter now)

Role	Opportunity	Timing
Tutorial/content authors	Chinese tutorial vacuum (this handbook is the first)	**Now** (biggest dividend)
Plugin developers	Directions the official repo lacks: TUI, remote access, mobile, cache tooling	**Now**
Tool-type projects	Speedup, latency visualization, MCP toolkits	Now (we validated with the speed-up plugin experiment)
Ecosystem infrastructure	Plugin marketplace, template generators, benchmark tooling	Mid term
Industry solutions	Finance/education industry plugin bundles	Mid term

11.6 Risks & uncertainty (honest edition)

1. **Breaking changes**: rc-stage API may change frequently — plugins/tutorials need ongoing follow-up (this handbook updates in sync)
2. **Ecosystem fragmentation**: plugin quality varies, standards missing — early stage needs "high-quality showcase projects" (this handbook + the Chapter 4 example are showcases)
3. **Official strategy shifts**: the official team may pull back directions (e.g., built-in TUI) — ecosystem projects need differentiated moats
4. **Model competition**: other models' iterations may weaken DeepSeek's appeal — but dsh's model-agnostic design (OpenAI-compatible endpoints) is a buffer
5. **Heat cooling**: if the dsh ecosystem doesn't take off, early investment may be "wasted" — but tutorials/skills themselves are transferable assets

11.7 This handbook's own outlook

- **Content**: continuously updated as dsh iterates (version-aligned with rc/stable releases)
- **Bilingual**: English edition catches up with Chinese
- **Cases**: more real industry cases (compliance-first)
- **Ecosystem**: link with awesome lists and official Discussions to become one of the ecosystem's content hubs

11.8 Final advice to readers

- **Want to try it**: install and play now (ch. 2) — cost is near zero
- **Want to invest**: pick a direction that is "missing from the official repo + matches your strengths" (TUI / industry plugins / tutorials)
- **Want to wait**: wait for the 0.1.0 stable, but you'll miss the early-ecosystem dividend — **early dividend ≈ patience of the first movers**

Predictions can be wrong, but "entering now costs the least" is probably not.

Chapter 12: Known Limitations & Boundaries — the Honest Edition

*Goal: list dsh's **known limitations** and usage boundaries without hype. When this handbook was written, dsh was still at `0.1.0-rc.6` (pre-release). These limitations are "facts of this moment", not "permanent fate" — they will change as versions iterate.*

All "known issues" come from hands-on testing + public feedback in the official repo discussions (deepseek-ai/deepseek-harness, as of 2026-08-13).

TL;DR (30-second version)

1. **The rc stage is the biggest uncertainty**: API/config can break at any time — pin your version line before investing
2. **Plugin ecosystem is early**: 60+ official packages but limited coverage; few third-party plugins; interactive forms (TUI) missing
3. **Cross-platform gaps**: real path/encoding bugs on Windows (UTF-16, 0x00)
4. **Performance boundaries unverified**: high concurrency, large codebases lack public stress data
5. **Official strategy risk**: open-source commitment vs commercialization is the black swan you must consider when choosing

12.1 rc-stage instability

Breaking changes happen anytime

dsh has gone through at least one **dependency breakage** from rc.1 to rc.6: inconsistent `@deepseek-ai/*` version lines caused plugins to fail installing (404/ERR_MODULE_NOT_FOUND). rc.6 has converged a lot, but **nothing guarantees rc.7 or the stable release won't break again**.

Real impact on you:

- Tutorials/plugins may need rewriting every couple of weeks
- Pinning the version line (`^0.1.0-rc.6`) is the baseline; but whether an rc-line upgrade is "backward compatible" or "start over" depends on the official team's mood

[!WARNING]
*Evaluate carefully before production use. At writing time the official team is iterating fast — **what works today may be deprecated tomorrow**.*

Known rc pitfalls (reproduced locally / in the community)

Pitfall	Symptom	Status
link-dev dependency breakage	`@deepseek-ai/*` ERR_MODULE_NOT_FOUND when	Root-caused (flat fallback dir mechanism); pin the dependency

	linking locally, but works via npm install	line to avoid
Windows path 0x00	workspace creation fails with Chinese/special-char paths (ENOENT)	Community-reported (#151/#396)
fs-sandbox race	post-check path check races with workspace-write	Community-reported (#159)
TUI missing	no interactive terminal UI in official examples (community builds exist, not merged)	Proposed (#392)

12.2 Early plugin ecosystem

Current state: 60+ official packages, but the ecosystem is overall at "day zero".

- **Few third-party plugins**: plugin threads in Discussions are exploding, but quality varies — most are first-plugin practice pieces
- **Missing interactive forms**: headless/JSON-RPC/Web are complete, but **no official example of terminal-native UI (TUI)** (the community deepseek-harness-tui proved it viable; not merged into official)
- **Docs lag the code**: official docs are architecture-focused; onboarding paths and pitfall records depend on community output (which is exactly why this handbook exists)
- **Version-line chaos**: multiple version lines coexist on npm; installing the wrong line = 404

Judgment: entering the ecosystem now is early-mover dividend (less competition, official willingness to support), but **don't expect "just follow the official docs and it runs"** — hitting pitfalls is the norm.

12.3 Cross-platform gaps

dsh is mature on macOS/Linux, but Windows support has a clear "second-class citizen" feel:

- **Path handling**: real bugs with Chinese/special-character paths (0x00 truncation, ENOENT); multiple community reports
- **Encoding**: UTF-16 handling has defects; occasional garbled terminal output
- **CLI/TUI debate unsettled**: there's an ongoing dispute in official Discussions about "CLI is the way vs TUI is the interactive future" (#386); investing in a terminal-UI form before the direction settles carries risk

12.4 Unverified performance boundaries

The handbook's benchmark covers **single agent, normal task scale**. These scenarios lack public data:

- **High concurrency**: parallel agents, large-scale task distribution

- **Large codebases**: context management and tool-call latency at million-line scale
- **Long-running stability**: hours-long sessions, memory-leak risk
- **Cache-hit boundaries**: the measured ~97% hit rate depends on "repetitive conversation patterns" — brand-new/cold-start tasks drop noticeably

12.5 Gaps vs mature agents

Compared with Claude Code, Codex, and other mature products, dsh's gaps are **structural** (not fixed by adding a few features):

Dimension	dsh (rc.6)	Mature agents
Ecosystem stage	day zero, few third-party assets	mature, complete
Out of the box	requires understanding profiles/plugin system	install & run
Stability	rc stage, breaking-change risk	stable releases, long-term compatibility promises
Docs	official is architecture-focused; onboarding via community	complete official docs + rich tutorials
IDE/toolchain integration	early	deep

dsh's positioning: not an "out-of-the-box car", but a "programmable LEGO base". Choosing dsh = choosing **freedom**, at the cost of **assembling it yourself**.

12.6 Official strategy risk

Black swans a selector must consider:

- **Open-source commitment**: MIT with no noise so far, but "official-grade plugin system" means the ecosystem is deeply bound to the official cadence
- **Model-binding tendency**: default experience favors DeepSeek models (any model via the llm plugin, but "official optimization" is DeepSeek-first)
- **Commercialization shift**: once dsh user numbers rise, the official team may launch managed services/paid tiers — whether the open-source edition keeps parity is unknown
- **Direction drift**: rc-stage features are added/removed frequently (the CLI/TUI debate is a signal); betting on the wrong direction = sunk cost

12.7 This handbook's own limitations

Finally, honestly about this handbook itself:

- **Written against rc.6**: every command, config, and data point verified on rc.6 — **newer versions may invalidate everything**

- **Limited benchmark sample**: single machine, single model, limited task set — not an authoritative evaluation
- **Chinese-first**: the English edition is a translation/condensation and may lag the Chinese content
- **Incomplete coverage**: dsh has 60+ official packages; this handbook deep-dives the core path; long-tail plugins aren't covered one by one

> ***Usage advice**: treat this handbook as an "rc.6 snapshot + methodology", not an "eternal bible". When versions update, run through the roadmap learning path acceptance checks first, then decide whether to update your dependency line.*

Info as of 2026-08-13 (dsh 0.1.0-rc.6). For newer versions, issues/PRs are welcome to correct.

Chapter 13: Security & the Sandbox Model

> ***Goal of this chapter:*** *Understand dsh's security skeleton — what the sandbox governs, how permissions are tiered, how approvals gate dangerous actions — and the real boundary cases the community has found.* ***This is the chapter that takes you from "it runs" to "safe enough for production."***

TL;DR (30-second version)

1. **Three layers of defense**: sandbox (where it executes) → permission presets (what it can do) → approval (whether dangerous actions need a human nod) — execution is *not* naked by default
2. **The sandbox only governs file effects**: `read-only` / `workspace-write` / `danger-full-access`; network and process visibility are outside its vocabulary (so cases like `taskkill` killing the host are "outside the vocabulary", not a sandbox breach)
3. **Permissions = two knobs bundled into presets**: `sandbox/mode` + `approval/policy`; two default presets: `workspace-write` (workspace writes + ask) and `danger-full-access` (no sandbox + never)
4. **Approval is fail-closed**: only `allowed-once` grants; ask with no responder = rejected (`unavailable`), never rejects deterministically — the default headless posture
5. **Community audits found real boundaries**: #159 fs-sandbox race, #584 scrubbedParentEnv collateral damage, #381 iframe clickjacking, #817 audit report (vm escape + unauthenticated local RPC)
6. **Enterprise baseline in one line**: default `workspace-write` + ask, keep loopback, install plugins only from trusted sources, keep sensitive directories out of the workspace

<details><summary>Chapter navigation</summary>

13.1 Security Design Philosophy: The Three Layers

dsh's security skeleton is three layers, each with one job (based on the official `docs/subsystems/*` architecture docs, verified: `sandbox.md` / `permission-presets.md` / `approval.md`; cross-checked against the hands-on measurements in Section 8.5):

Layer	What it governs	Official packages	One-liner
Sandbox	Where code runs, which files it can touch	`sandbox/sandbox`, `sandbox/sandbox-local`, `sandbox/sandbox-policy`	Isolates the file effects of command execution
Permissions	What the agent may do	`interaction/permission-presets`	Bundles sandbox + approval into named presets
Approval	Whether dangerous actions need a human confirm	`interaction/user-approval`	The last gate before least privilege

Least privilege is the shared design intent across all three layers: the default preset only grants "workspace writes + ask before critical operations"; any escalation (switching to `danger-full-access`, changing the approval policy) happens explicitly and is written to the session log — the `permission/preset` event is a **log-only record of user intent**, kept out of the model transcript, so you can always look back at "who raised privileges, and when."

Threat model in one line: the three layers defend against "untrusted input driving the agent across its boundaries" — prompt injection, malicious plugins, induced high-privilege operations (#381 iframe hijacking is a direct shot at this chain). They do *not* promise to defend against "trusted code going rogue" (a plugin *is* executable code, see 13.5), nor against network/process-level attacks.

> *Core takeaway: **dsh's tool execution has isolation and approval layers by default — it is not naked execution.** But it is not a "security operating system" either — the sandbox doesn't block network or processes, and approval can be self-answered by the model (13.6).*

13.2 Sandbox Mechanics: FS Boundaries and Tool Sandboxing

The official sandbox doc is explicit: **the sandbox only governs file effects (filesystem effects)** — network and process visibility are not part of its vocabulary.

The three modes (`SandboxMode`):

Mode	Allowed file effects	Notes
`read-only`	Read-only + required sinks (e.g. `/dev/null` on POSIX)	Windows ACL backend has no explicit writable root → reports `partial`
`workspace-write`	Workspace root + backend-promised temp areas writable	**Default preset**; root comes from the session's immutable cwd
`danger-full-access`	Unrestricted	Spawns the original argv directly, bypassing `ctx.sandbox`

Execution details:

- **Policy is per-call**: `SandboxExecutionPolicy` (mode + workspaceRoot + sessionId) is resolved at each capability call rather than being pinned to a provider — the same provider can serve a `read-only` bash and a subagent that needs a writable state directory
- **Workspace root canonicalization**: first canonicalize per filesystem semantics (`symlink/..` resolved to real directories), then apply lexical normalization — prevents symlink-based boundary escapes
- **Backend matrix**: Linux bwrap/Landlock, macOS Seatbelt, Windows ACL restricted-token; `enforcement: full | partial` — old Landlock ABIs and the Windows ACL Everyone/hard-link boundaries are the current `partial` cases (**when the promise is watered down, consumers must treat it as `partial`**)
- **FS tools share the boundary**: file tools like `write`/`edit` are held to the same `workspace-write` constraint — it's not "file tools pass through, only bash is sandboxed" (#149's recursive workspace deletion happened under `workspace-write`)
- **Writes restricted, reads not**: the current permission model constrains **writes**; content outside the workspace can still be read (#492 raised this when proposing an evaluation isolation mode)

Tool sandboxing: `bash`/`pwsh` go through `bash-sandbox` / `pwsh-sandbox` (consuming `ctx.sandbox`) and spawn child processes inside the sandbox. On Windows, the pwsh sandbox has ACL constraints (we hit temp-permission issues in Chapter 8's hands-on testing; #758 also reports that cleaning up the Windows sandbox temp directory can permanently crash it).

Related family cases: #159's race is a TOCTOU (the path is swapped between check and use); #278 is a different widening trick — when `/tmp` is the workspace, a restricted child process can use rebinding to widen the `workspace-write` grant. Shared lesson: **choose a trusted location for the boundary root, and never rely on a single path check alone.**

13.3 Permission Model: Tool Permission Tiers

The official permission presets bundle **two knobs** — `sandbox/mode` and `approval/policy` — into named presets, surfaced to the client as a single "Permissions" selector. Two default presets:

Preset	sandbox/mode	approval/policy	Semantics
`workspace-write`	workspace-write	ask	Writable inside workspace, asks before operations (default)
`danger-full-access`	danger-full-access	never	No sandbox, no asking
`custom` (derived state)	any combination	any	The "non-preset" state after manually turning the knobs; cannot be a switch target

Tool permission tiers (ordered by blast radius; "suggested preset" per tool is derived from the official docs, not tested tool-by-tool — **inferred, pending verification**):

Tier	Representative tools	Suggested preset	Risk	Notes
Read-only probing	`read/grep/glob`	read-only compatible	Low	Read-only, nothing written; reads not limited to the workspace
Workspace writes	`write/edit/str_replace_editor`	workspace-write	Medium	Write boundary = workspace root + temp areas
Command execution	`bash/pwsh`	workspace-write (sandboxed)	High	The sandbox doesn't block network or processes
Full access	any (switch to danger-full-access)	danger-full-access	Extreme	All gates off

> *The official docs are explicit that the sandbox **only governs files** — so the high risk of the "command execution" tier can't be sandboxed away. It must be backstopped by approval and trust boundaries (13.5 / 13.7).*

Real boundary cases (details in 13.6): recursive deletion of the entire workspace with zero confirmation under `workspace-write` (#149); `danger-full-access` accidentally deleting an entire home directory (#461, real incident); on Windows, the minimal preset allows out-of-workspace writes with no approval (#523); an agent inside the sandbox can `taskkill` the host (#466 — process visibility is outside the sandbox vocabulary).

13.4 Approval Flow: From Request to Grant

The official approval doc: approval answers one question — "can this specific action happen?"

Fail-closed result set (`ApprovalOutcome`):

Outcome	Meaning	Effect on the caller
`allowed-once`	One-time grant (only for the action asked about)	✅ proceed
`rejected`	Explicit denial	❌ abort
`cancelled`	Request withdrawn (AbortSignal)	❌ abort
`unavailable`	No responder / responder errored	❌ abort (fail-closed by default)

Policy (`ApprovalPolicy`): `ask` (default — handed to the responder chain; empty chain → `unavailable`); `never` (deterministically rejects everything, dispatches no responders — the strict headless/CI posture).

Flow: tool call → `approval/request` waterfall (plugins can intercept/rewrite) → responder (a human in the Web UI, or a one-shot machine decision via the ACP bridge) → every request writes a paired audit pair `approval/asked` + `approval/decided` (same `ApprovalRequestId`).

Request content (`ApprovalRequest`): agent (a responder only answers for the agents it owns) + toolName + callId (linked to the already-streamed tool call, so parameters aren't re-rendered) + reason (why the question is being asked) + signal (AbortSignal for withdrawal).

Three known pitfalls:

1. **Approval loop**: #250 reproduced "model self-approving danger-full-access" for real (the Web approval channel can be driven by the model itself) — **approval is not a security boundary**, only a human-in-the-loop gate
2. **Concurrent misrouting**: #453 — clicking Allow during concurrent sandbox approvals aborts a *different* call (UI response and callId mismatched)
3. **Reconnect drops pending approvals**: #646 — silent loss of pending approvals on reconnect (resync clears them + replay race); headless approval behavior is undefined (#291)

13.5 Plugin Security Audit Checklist

Plugins are dsh's capability boundary ("everything is a plugin", see Chapter 4) — **installing a plugin = installing executable code.** The checklist distilled from community audits (#817 and family):

#	Check	Community basis	Disposition
1	Is the source trustworthy? `dsh plugin add` has no signature/source validation	#587	Install only from official/well-known sources; read the source before installing
2	Minimal boot-time permissions? Plugins get write access to the whole config tree at boot	#587	Beware plugins that "auto-edit your config after install"
3	Not treating the vm as a security boundary? Workflow/dynamic-plugin vms can escape	#243 #451 #774 #778	Treat the vm as an isolation engine, not a security layer
4	Child-process env scrubbed correctly? `scrubbedParentEnv` substring-matching hits legitimate variables	#584	Check whether variables containing the KEY substring (e.g. KEYBOARD/MONKEY) get wrongly scrubbed (**mechanism inferred from the thread title**)
5	Secret redaction not fail-open? `settings role('secret')` has redaction gaps	#226	Don't put secrets into prompts; verify the redaction logic
6	Security hooks not silently disabled? `timeout: 0` fails open	#460 #583	Set positive hook timeouts; loading failures must surface as errors
7	Audit trail? `approval/asked/decide`	official approval.md	Critical operations traceable per session

	d pairs are queryable		
8	No plaintext session leakage? `.tmp` plaintext session residue after crashes	#674	Disable crash dumps in sensitive environments / clean up periodically

13.6 Known Security Cases from the Community

All 4 threads below verified to exist via `gh api graphql` (deepseek-ai/deepseek-harness Discussions, 2026-08-14):

Thread	One-liner	Type	Impact
#159 *(https://github.com/deepseek-ai/deepseek-harness/discussions/159)*	`fs-sandbox` post-check pathname race can bypass the `workspace-write` file boundary	Sandbox race	TOCTOU: the path is swapped between check and use, enabling out-of-bounds writes
#584 *(https://github.com/deepseek-ai/deepseek-harness/discussions/584)*	`scrubbedParentEnv` substring-matching wrongly scrubs legitimate env vars like KEYBOARD/MONKEY (cherry-pickable fix included)	Child-process env	Legitimate env vars get scrubbed → behavioral anomalies
#381 *(https://github.com/deepseek-ai/deepseek-harness/discussions/381)*	Default localhost web UI can be clickjacked via cross-site iframe, inducing a `Full access` grant and driving high-privilege operations	Frontend clickjacking	Malicious webpages trick users into clicking out high privileges in the dsh UI
#817 *(https://github.com/deepseek-ai/deepseek-harness/discussions/817)*	Security audit report: sandbox/approval boundary bypasses + unauthenticated local RPC (PoC available privately)	Systematic audit	Covers vm escape (family #243 #778), unauthenticated local `/api` (family #451, CVSS 8.8), approval loop (#250)

High-value family supplements (thread numbers from `docs/research/discussion-mining.md` §2.6 in this repo; that report says all 780 threads were programmatically verified):

Thread	One-liner	Lesson
#149	Whole-workspace recursive deletion with zero confirmation under `workspace-write`	Workspace contents also need protection from "self-destruction"
#250	Web approval loop: model self-approving danger-full-access	Approval is not a security boundary
#461	Full Access mode deleted an entire home directory (real incident)	High-privilege preset risk, proven in the wild
#587	Third-party plugins get whole-config-tree write access at boot	Plugin trust = supply-chain security
#466	An agent inside the sandbox can `taskkill` the host	Process visibility is outside the sandbox vocabulary
#674	`.tmp` plaintext session residue after crashes is never cleaned	Privacy risk

Disclaimer: these are community reports from the rc.6 era (within 48h of the 2026-08-13 release); some may already be fixed in newer rcs — keep the version context in mind when citing.

13.7 Enterprise Security Baseline

Decision guidance for "going to production / rolling out to a team" (aligned with Chapter 6's style):

Dimension	Baseline	Basis
Network exposure	**Keep loopback** (official rejection of `--host 0.0.0.0` is intentional); don't bypass until remote auth matures	#76 #130 #397
Permission preset	Default `workspace-write` + ask; `danger-full-access` only on isolated machines / one-shot tasks	official preset table + #461
Approval policy	Human responder chain required; headless uses `never` (deterministic rejection) with a review gate in front	approval.md + #291
Plugin governance	Whitelisted sources + run the 13.5 checklist before installing; ban bare `dsh plugin add github:`	#587 #656
Workspace boundary	Sensitive directories (home/keys) stay out of the workspace; workspace contents go under version control regularly	#149 #461
Audit	Session logs retained (approval pairs traceable); guard against `.tmp` plaintext residue	approval.md + #674
Upgrade discipline	Re-run the 13.5 checklist on every rc upgrade (sandbox backends / preset table can change)	Chapter 12 rc-integrity stance

Decision memo: local personal dev → default preset is enough; team-shared → `workspace-write` + ask + plugin whitelist; CI/unattended → headless + `never` + code review gate in front; don't run `danger-full-access` long-term anywhere just for convenience.

Scenario → recommended combination (decision guidance, derived from the baseline table above):

Scenario	Recommended combination	Why
Local personal dev	Default `workspace-write` + ask	Works out of the box; approval isn't annoying
Team-shared host	workspace-write + ask + plugin whitelist	Multi-user; plugin supply-chain risk is amplified
CI / unattended	headless + `never` + review gate in front	Deterministic rejection; nothing hangs waiting on a human
Isolated machine / one-shot task	`danger-full-access` (the machine itself is isolated)	Maximum authorization, but blast radius capped at one box

Hands-on exercises

1. **Understand**: what file effects does each of the three sandbox modes allow? Why is "network and processes are outside the sandbox vocabulary"?

Check yourself: Section 13.2's three-mode table and the "backend matrix" paragraph

2. **Understand**: why is `never` under headless the "strictest" rather than the "loosest"?

Check yourself: Section 13.4's "Policy" paragraph (deterministic rejected)

3. **Hands-on**: open `dsh web`, look at the two presets in the Permissions selector; switch to `danger-full-access` and back, and watch the `permission/preset` event in the session log

Check yourself: Section 13.3's preset table

4. **Hands-on**: before installing a community plugin, walk it through the 13.5 checklist item by item (source/source code/permissions/audit) and write the conclusion into your notes

Check yourself: Section 13.5's checklist table

5. **Think**: why does #250's "model self-approval" prove that approval is not a security boundary? If approval can't stop it, what does an enterprise fall back on?

Check yourself: Section 13.6 + the 13.7 baseline table

FAQ

- **Q: Can the sandbox stop malicious code?** No — don't use it as a security boundary. The sandbox only governs file effects; network (e.g. SSRF) and processes (e.g. #466's `taskkill`) are outside its vocabulary. Defending against malicious code relies on plugin trust (13.5) and network/process-level isolation (containers/VMs/remote sandboxes — the official docs are explicit these are sibling implementations of the capability seam, not `ctx.sandbox` providers).
- **Q: Why is `never` under headless actually safer?** `never` isn't the lax "never ask" mode — it's **deterministic rejection**: any approval request is directly `rejected`, with no responders dispatched — unattended runs never "hang waiting for someone to click Allow." Actions that need to happen rely on an upfront review gate or explicit configuration, not runtime questions.
- **Q: So is `workspace-write` safe?** Relatively safe inside the boundary — but #149 shows "recursively deleting the whole workspace" gets zero confirmation under `workspace-write`. The sandbox stops *boundary crossing*, not *self-destruction inside the boundary*. Put important code under version control.
- **Q: Chapter 12 already mentioned #159 — why cover it again here?** Chapter 12 was a "known-issues quick table" (one-liner + status); this chapter expands the mechanism (how the post-check race

happens), the family (#278 /tmp rebind), and the mitigations (don't trust a single path check, keep critical directories out of the workspace).

- **Q: Can these security thread numbers be trusted? Surely someone could have made them up?** The 4 core threads (#159 #584 #381 #817) were each verified to exist via `gh api graphql` (links in the 13.6 table); the family supplements come from `docs/research/discussion-mining.md` in this repo (780 threads fetched with full pagination and programmatically verified).

Next chapter: Chapter 14: Caching & Cost Engineering

Chapter 14: Caching & Cost Engineering

> ***Goal of this chapter:*** *Turn "it's cheap" from folklore into engineering — the conversation-cache mechanism, measured hit rates and how to improve them, the cost model, reasoning-tier interplay, real-task budgeting, and how to see where every token goes.*

TL;DR (30-second version)

1. **Cache is the #1 cost variable**: DeepSeek's context cache bills repeated input at a **98% discount (Flash) / 99%+ discount (Pro)** — agent workloads are input-heavy, so the hit rate directly determines the cost scale
2. **Measured hit rate in this handbook: 97%**: session continuity + stable tool schemas + the nature of multi-step agent workloads (Section 5.1.1); community long-run measurements reach 99.7% (#560 *(https://github.com/deepseek-ai/deepseek-harness/discussions/560)*)
3. **Three hit-rate principles**: keep the session alive (don't keep creating new ones), keep the prefix stable (don't keep changing config), interrupt less (write clear acceptance criteria once)
4. **Two independent levers**: reasoning tier controls *how much thinking*, cache controls *the input unit price* — they stack, so `low` + high hit rate = the cheapest combination
5. **Cost must be visible**: the session stats line shows cache hit %; per-turn token counts are awaiting official implementation (#735 *(https://github.com/deepseek-ai/deepseek-harness/discussions/735)*) — until then, use the stats line + a community cost-tracker plugin

<details><summary>Chapter navigation</summary>

14.1 How Conversation Caching Works

DeepSeek's API **context cache** (prompt caching): repeated/similar input tokens are billed at the **cache price** instead of full price. dsh's input for every model request = system prompt + skill catalog + conversation history + tool schemas (Section 8.3) — **the stable prefix portion is the natural cache candidate**, which is why agent workloads have far higher cache potential than ordinary chat.

Billing rules (Section 5.1.1 pricing table):

Charge	Miss price	Cache-hit price	Discount
Input (Flash)	$0.14/M	$0.0028/M	**98%**
Input (Pro)	$0.435/M	$0.003625/M	**99%+**

Input structure per request (Section 8.3) — determines which tokens can hit:

```
system prompt (stable) + skill catalog (stable) + conversation history (incremental) + tool
schemas (stable) + this turn's messages (incremental)
      └──────────────────────── cache-hit zone (repeated) ────────────────────────┘   └── miss
zone (new increments) ──┘
```

The bigger the hit zone and the smaller the increments → the higher the hit rate. Once you understand this structure, the three optimization levers in 14.3 are all really "make the hit zone bigger."

Cache isn't just cheaper, it's faster: cold start with context injection takes ~110s on the first turn; a hot cache takes ~1s (measured in Chapter 1) — first-token time drops by an order of magnitude on hits.

> *Detail note: hits depend on "repeated/similar prefix"; the exact matching rule is decided API-side (inferred, pending verification). In practice, "keep the prefix stable" is enough to reliably capture the discount.*

14.2 Measured Hit Rate: Where 97% Comes From

This handbook measured a cache hit rate of up to **97%** in real dsh session stats (Section 5.1.1). Why dsh's hit rate is unusually high:

Reason	Explanation
Session continuity	Multiple turns in one session repeat the system prompt / skill catalog / history → hits
Stable tool schemas	Every step carries the same tool definitions → hits
Agent workload nature	Multi-step tool chains repeatedly carry the same context → naturally high hit rate

Boundary (honest edition): 97% depends on "repetitive conversation patterns" — **brand-new tasks / cold starts drop noticeably** (Section 12.5). Community long-run measurements reach **99.7%** (#560 *(https://github.com/deepseek-ai/deepseek-harness/discussions/560)*), confirming "longer sessions → higher hit rates."

How it was measured: the "cache hit %" in the session stats line at the bottom of the Web UI (Section 6.6); the comparison baseline is 5.1.1's "50-step tool chain task, 2.4M input tokens" — a typical sample of the input scale.

Metric	Where to look	Behavior on hits
Cache hit %	Session stats line (bottom of Web UI)	Stays high (97%+ in long sessions)
First token	Per-turn end line	Drops by an order of magnitude from 10s+ to under 1s (FAQ Q4 in Chapter 6)
Total tokens	Session overview	Small increments = the hit zone is growing (14.1 structure)

14.3 Optimizing the Hit Rate in Practice

Three levers, ordered by cost-effectiveness:

Lever	What to do	Why it works
Session continuity	Keep long tasks in one session; avoid creating new sessions frequently; batch tasks in the same session/prefix	Repeated history messages → hits
Stable prompts	Keep the system prompt / skill catalog prefix stable; don't churn config; keep tool schemas unchanged	Identical prefix → hits
Fewer interruptions	Write clear acceptance criteria once (Section 5.7 / the lesson in Chapter 10's cases) instead of "reopening sessions and re-explaining context"	Avoids rebuilding the prefix → full-price restart

Grounding check: when the hit rate drops below expectations, the first thing to suspect is "did the prefix change?" — changing config, switching tiers, or touching the skill catalog all invalidate the cache (Section 6.6 monitoring line).

14.4 Cost Model: How the Hit Rate Drives Your Bill

Cost model in one line (Section 6.6): `total cost ≈ output tokens × output price + missed input × miss price + hit input × hit price` — agent workloads are input-heavy, so **the cache hit rate is the #1 cost variable**.

Using Chapter 5's Flash input prices, here's a worked example (**10k input tokens**, hit-rate comparison). Arithmetic sample (90% hits): `9,000 × $0.0028/M + 1,000 × $0.14/M ≈ $0.000165` — split hits from misses first, then multiply each by its own price:

Hit rate	Hit tokens	Miss tokens	Input cost (Flash)	Relative cost

90%	9,000	1,000	≈ $0.000165	1×
50%	5,000	5,000	≈ $0.000714	≈ 4.3×
10%	1,000	9,000	≈ $0.001263	≈ 7.6×

Scaled to 1M input tokens: 90% hits ≈ **$0.0165**, 10% hits ≈ **$0.126**, full price $0.14 — **nearly an order of magnitude apart**.

Scaled again to this handbook's measured scale (5.1.1: 50-step tool chain, 2.4M input tokens): 97% hits ≈ **$0.017** vs full price ≈ **$0.34** — roughly **20×**. That's the numerical source of 5.1.1's "cost far below the naive miss-price estimate."

> *Note: this table only covers the input side — Chapter 5's pricing table only records input prices; for the output side, check the official pricing page. In input-heavy agent scenarios, the input side is the dominant variable.*

14.5 Reasoning Tiers and Caching: the Cost Matrix

Reasoning tiers control "how much thinking," and cache is an **orthogonal lever**: tiers affect total token count, cache affects the input unit price, and they multiply. Fewer thinking tokens → lower total tokens (Section 6.6), and the context-repeated portion enjoys the cache discount too (inferred, pending verification).

Tier	Thinking tokens	Relative cost	Best for (Section 6.2)	Cost strategy
`low`	Fewest	Lowest	Simple/deterministic turns: file ops, batch jobs, cheap steps in a tool chain	Default tier for batch jobs
`high`	Medium	Medium	Everyday agent tasks (default)	Default quality/cost balance
`max`	Most	Highest	Complex reasoning, long-chain planning, debugging	Fine for one-off complex tasks; don't run it in batch loops

How the levers combine:

- **Long tool-chain tasks (20–50 steps) benefit the most**: the cumulative effect of downgrading thinking per step is significant (FAQ Q2 in Chapter 6) — both levers (tier × hit rate) act on every step
- Simple turns at `low` show almost no quality difference; complex reasoning at `low` may skip critical steps (FAQ Q3 in Chapter 6)
- **Stacked example** (2.4M-input-scale task, Flash prices): `low` + 97% hits ≈ under $0.017 (fewer thinking tokens → actually less; inferred, pending verification); `max` + repeated cold starts ≈ over $0.34 — **the same task is 20×+ apart with both levers fully on vs fully off**

- Note: `low/high/max` are the tier names of the gateway this handbook measured on (pi-ai/opencode-go); the default deepseek-official adapter uses `off/high/max` (Section 6.2 note)

14.6 Budget in Practice: Cost Estimate for Case A

Budget using Chapter 10's **Case A (data-quality analysis, 186s)** as the sample: tool chain `read → write(clean.py) → bash → write(visualize.py) → bash → read → summarize`, ~6–8 steps. Scaled from the 5.1.1 magnitude (50 steps ≈ 2.4M input tokens), Case A's input is roughly **500k tokens (inferred, pending verification)**:

Scenario	Hit rate	Input cost (Flash, estimated)
Session continuity + stable prefix (recommended)	97%	≈ **$0.0035**
New session mid-way / prefix changed	10%	≈ $0.063
Extreme: full price every time	0%	$0.07

Step-level breakdown (why session continuity saves money) — only the first turn is a full-price cold start; every later turn misses only its small increment of "this turn's messages + tool results" (inferred, pending verification):

Step	Content	Input tokens (inferred, pending verification)	Hit status
1	read(sales_data.json)	~50k	Cold start (miss)
2–6	write/bash/write/bash/ read/summarize	~450k	Prefix hit zone, tiny increments

Conclusion: a real 3-minute task costs under **1 cent** on the input side; the same work, done differently (session continuity vs repeated cold starts), differs by **~18×**. The output side (two scripts + summary) is orders of magnitude smaller than the input, so total cost stays in the single-digit cents (inferred, pending verification). This is the per-task source of "dsh with V4-Flash costs about 1/10 to 1/30 of Claude" (measured in Chapter 1).

14.7 Measurement: Making Token Usage Visible

What exists today:

- Session stats line at the bottom of the Web UI: `N turns · M steps | LLM Xs · Tool calls Ys | Avg first token ...` (Section 6.3)
- Per-turn end line: `10:14 · took 9m34s · first token 1.7s · 79 tok/s` (#735 *(https://github.com/deepseek-ai/deepseek-harness/discussions/735)* original-thread screenshot)
- Session overview shows total-token stats; the stats line shows "cache hit %" (Section 6.6)

The gap: **per-turn token counts**. Official discussion #735

(https://github.com/deepseek-ai/deepseek-harness/discussions/735) (filed 2026-08-14, title "【友好显示】希望在每轮对话中加入本轮对话 token 消耗量" / "Friendly display: add per-turn token usage to each turn", 2 comments) is exactly this request — confirmed to exist in the thread; not yet implemented officially.

Community suggestion (already posted to #735's comment thread): the per-turn display should split into **two numbers** — ① total tokens this turn (quick glance at spend); ② hit/miss tokens this turn (see the optimization headroom). Looking at total tokens alone misleads: the same 10k tokens costs 7.6× more at a 90% vs 10% hit rate (see 14.4).

Interim tooling: the community `dsh-plugin-cost-tracker` (Sections 9.5/9.6; real-time token-cost tracking, plugin/MCP form) — implemented through Chapter 4's plugin extension points, tallying `usage` on request/session events (inferred, pending verification); or reconcile manually from session logs / the API usage field. Chapter 11 predicts "cache-hit-rate tooling" is the mid-term theme (11.2) — once #735 lands, the cost-transparency loop closes.

14.8 Cost Pitfall Checklist

#	Pitfall	Symptom	Fix
1	Misreading cache hits as "the model got faster"	A/B comparison skewed by 1s vs 110s (Chapter 6 pitfall #6)	A/B tests must use a **fresh prompt**
2	Creating new sessions constantly burns money	Every task cold-starts; input billed at full price	Keep long/batch tasks in one session
3	Config changes silently kill the hit rate	Hit % suddenly drops; cost climbs	Monitor the hit % in the stats line; keep the prefix stable
4	Judging cost by total tokens alone	10k tokens at 90% vs 10% differs by 7.6×	Split hit/miss and look at both (#735 suggestion)
5	Budget missing the output side	Input-only math doesn't add up	Add output prices from the official pricing page (Chapter 5 only lists input prices)
6	Tier roulette	`max` on batch jobs doubles the cost	Use `low` for batch, `max` only for complex tasks

14.9 Decision Recommendations at a Glance

Scenario	Recommendation
Long tool-chain tasks (20–50 steps)	`high` + session continuity + stable prefix; 2.4M-input scale ≈ $0.017 (97% hits)
Batch / simple turns	`low` + batched in the same session (biggest hit bonus)
Complex reasoning / debugging	`max` (quality over cost); cold start is acceptable for one-off tasks

Cost-sensitive batch processing	`low` + headless + serial in one session
Performance/cost comparison tests	Fresh prompt + fixed tier + median of multiple runs (Sections 6.4/6.6)
Hit rate dropping abnormally	Check whether the prefix changed first (config / skill catalog / model tier)

Hands-on exercises

1. **Understand**: why is "cache hit rate the #1 cost variable"? Use the 14.4 10k-token table to explain how many times more 90% costs vs 10% hits

Check yourself: the 14.4 comparison table (7.6×) and the 6.6 cost-model one-liner

2. **Hands-on**: run a 3-step task and watch the session stats line at the bottom of the Web UI and the per-turn end line (time / first token / tok/s); estimate how much of this task hit cache

Check yourself: the 14.7 measurement methods and the 6.3 stats-line format

3. **Hands-on**: run the same batch of tasks two ways — "a new session per item" vs "all in one session" — and compare wall-clock times; the new-session version will almost certainly be noticeably slower on the first turn (cold start)

Check yourself: 14.1's measured 110s cold start vs 1s hot cache

4. **Think**: why does "looking only at this turn's total tokens" misjudge cost? Which two numbers did the #735 comment thread suggest splitting the per-turn display into?

Check yourself: the "Community suggestion" paragraph in 14.7

5. **Hands-on**: using #735's suggested metric, manually split one session's total tokens into "hit/miss" parts, price them with 14.4's unit prices, and verify the "order of magnitude" claim

Check yourself: the 14.4 comparison table and the 14.2 measurement table

FAQ

- **Q: What's the highest possible cache hit rate?** 97% measured in this handbook (Section 5.1.1); community long runs reach 99.7% (#560 *(https://github.com/deepseek-ai/deepseek-harness/discussions/560)*). But **brand-new tasks / cold starts drop noticeably** (Section 12.5) — the hit rate is a function of *how you work*, not a constant of the model.
- **Q: Does the `low` tier noticeably hurt quality?** Almost no difference for simple deterministic tasks (file ops, batch processing); complex reasoning, long-chain planning, and debugging at `low` can miss critical steps (FAQ Q3 in Chapter 6). Recommended: `high` for everyday, `low` for batch/simple tasks.

- **Q: When will the official per-turn token display ship?** #735 *(https://github.com/deepseek-ai/deepseek-harness/discussions/735)* is a feature request filed 2026-08-14 (2 comments) — **not implemented yet**. Until then: the session stats line's hit % + the community `dsh-plugin-cost-tracker` (Chapter 9).
- **Q: Do third-party gateways / self-hosted providers get the cache discount too?** The cache discount is a DeepSeek API-side capability (Chapter 5 pricing table); third-party gateways' cache support and billing must be checked on their own pricing pages (inferred, pending verification).
- **Q: Why does a task get noticeably faster after a cache hit?** The hit prefix doesn't need recomputing, so first-token time drops by an order of magnitude (cold start ~110s vs hot cache ~1s, measured in Chapter 1). Tell-tale sign: first token suddenly drops from 10s+ to under 1s — almost certainly a hit (FAQ Q4 in Chapter 6). Use fresh prompts for performance comparisons to avoid cache interference (Chapter 6 pitfall #6).

Info as of 2026-08-14 (dsh 0.1.0-rc.6). Pricing and cache policy are subject to the official docs; items marked "(inferred, pending verification)" are extrapolations or estimates — corrections via PR after real-world testing are welcome.

Appendix · 附录（中文原文）

The appendices below are maintained in Chinese; they are included here in their original form so the PDF stays complete. English readers can find the equivalent concepts inside Chapters 1–14.

Appendix A · appendix-glossary (Chinese original)

附录 A：术语表与命令速查

术语表

术语	一句话解释
Harness	套在模型外的工程层：会话、工具、上下文、循环控制
dsh	DeepSeek Harness 的命令行名（dsh 命令）
Profile	一种可启动形态（web / headless / 自定义），= bundle 栈 + patch 层
Bundle	一组插件的集合（官方：dsh-base、dsh-web-app、dsh-headless）
Cordis	dsh 底层的插件容器（依赖注入、事件、生命周期）
插件（Plugin）	cordis 插件；可同时携带 host 半（Node）与 client 半（浏览器）
host 半 / client 半	插件在 Node 侧（工具/服务/事件）与浏览器侧（UI）的两副面孔
扩展点	官方提供的钩子（agent/request、settings、conversationEvents、slots）
agent/request waterfall	每步模型请求前可改配置的事件链（插件在此注入 reasoningEffort 等）
reasoning_effort	思考强度（官方适配器 off/high/max；low 为实测网关档位）——速度与质量的权衡旋钮
headless	一次性 CLI 任务模式（`dsh --profile headless "任务"`）
compaction	长对话上下文压缩
subagent	子代理（并行委派任务）
MCP	模型上下文协议（接入外部工具服务器）
workflow	多步确定性工作流编排
locations	工具返回的文件路径（驱动产物追踪）
extension point	官方钩子（agent/request 等），插件接入点
waterfall	事件链：监听者可改配置传给下一个（agent/request

	用此注入）
context cache	DeepSeek 提示词缓存（重复输入按折扣价计费）
缓存命中率	输入 token 走缓存价的比例（dsh 实测可到 97%）
TUI	终端 UI（官方未内置，需插件）
turn	对话的一轮（用户+助手+工具调用）
step	turn 内的一个推理步骤
guard	循环卫生/工具超时插件
skill	技能（skill-catalog 注入上下文，模型按需调用）
inject	cordis 依赖注入（插件声明所需服务）
产物文件	模型创建/修改的文件（对话末尾的可打开 chips）
sandbox	命令执行的隔离沙箱
rc	预发布版本（0.1.0-rc.6），迭代快、可能有破坏性变更

命令速查

dsh 核心

```
dsh web                                    # 启动 Web UI
dsh --profile headless "任务"              # 一次性任务（脚本/CI）
dsh --version                              # 版本
dsh --dump-config                          # 合成配置树
dsh plugin --profile <name> add <pkg>      # 安装插件
```

环境

```
node --version                             # 需 ≥ 22
npm install -g @deepseek-ai/dsh            # 全局安装
npx -y @deepseek-ai/dsh web                # 免安装运行
```

排障

```
netstat -ano | findstr 3080                # 端口占用（Windows）
taskkill /PID <pid> /F                     # 杀进程
cat ~/.dsh/settings.yaml                   # 全局配置（模型/推理档位）
```

插件开发

```
# 挂载到 profile（web 为例）
# ① package.json 加依赖："pkg": "link:C:\\path\\to\\pkg"
# ② cordis.patch.yml 加：
#    - insert:
#        - id: <插件 id>
#          name: <npm 包名>
cd ~/.dsh/profiles/web && pnpm install          # 安装
```

配置参考（settings.yaml）

```
agent-default-model:
  model: deepseek-v4-flash     # 或 deepseek-v4-pro
  reasoningEffort: high        # off（关闭思考/最快）/ high（默认）/ max（最强）
```

附录 B：官方包速查大全 · **附录 C**：同模型多 Agent 实测

Appendix B · appendix-packages (Chinese original)

附录 B：官方包速查大全（@deepseek-ai/*）

本清单为「已知包」清单：*收录白皮书正文实测/提及过、且能在 npm registry 上核实的官方包（@deepseek-ai/dsh-* 与 @deepseek-ai/cordis），包描述来自 npm view 实测结果。dsh 处于 0.1.0-rc 快速迭代期，**包名与描述随版本更新**——若与官方仓库 packages/AGENTS.md 或 npm 最新描述不一致，以官方为准。未能核实的包标注「（待补全）」，不虚构包名。*

包数口径：*本清单收录经 npm 核实的 **33 个核心包**（CLI 直接依赖 53 个 + 家族发布总量 221 个，见 08 章 8.1 节 说明）；白皮书/官方所称「60+ 能力包」指仓库 packages/ 下全部子目录，完整列表见官方仓库 packages/README.md（47 组权威表）。*

包名与仓库布局对应关系：npm 包 @deepseek-ai/dsh-xxx ≈ 官方仓库 packages/<group>/xxx（如 dsh-tool-fs ↔ fs/tool-fs）。

验证方式：npm view @deepseek-ai/<name> description（2026-08 快照）。

能力域取值：工具（模型可调用的工具/执行管道）· 上下文（请求上下文组装与压缩）· 会话（会话持久化/标题/遥测）· 子代理（委派子任务）· MCP（外部工具服务器接入）· 工作流（多步编排）· 安全（沙箱/审批/循环卫生）· LLM（模型接入与策略）· UI（浏览器界面与客户端服务）· 核心（CLI/骨架 bundle，不属上述能力域）。

核心（Core）：CLI 与骨架 bundle

dsh 的"地基"：装一个 @deepseek-ai/dsh 就能跑，profile 由内置 bundle 栈堆出来（dsh-base → dsh-headless / dsh-web-app）。

包名	用途	能力域
@deepseek-ai/dsh	dsh CLI：profile 启动、插件管理、浏览器 UI 别名（npx -y @deepseek-ai/dsh web）	核心
@deepseek-ai/dsh-base	共享 dsh 核心 profile bundle：每个 profile 的第一层补丁，在空 profile 根上插入基础插件行	核心
@deepseek-ai/dsh-headless	headless 一次性 bundle：无 Host/HTTP/浏览器层的直接 core Agent/Session 运行器	核心
@deepseek-ai/dsh-agent	Agent 接口、注册表、发起者作用域与事件词汇（插件开发常直接依赖）	核心
@deepseek-ai/dsh-web-app	dsh 浏览器面 bundle：web patch 层（前端 dist 服务、web 面提示词、bash 运行时变量、URL 行）	核心
@deepseek-ai/cordis	底层元框架：现代 JavaScript 应用的依赖注入/事件/生命周期容器（dsh 的插件底座）	核心

工具（Tools）：模型可调用的能力

模型实际看到的工具名是**简短动词**（read/write/grep/glob/edit/bash/web_search/todo_write…），底层由本组包提供实现。

包名	用途	能力域

@deepseek-ai/dsh-tools	工具注册表与执行管道（tool registry + execution pipeline）	工具
@deepseek-ai/dsh-fs	抽象文件系统能力 seam（ctx.fs）：词汇类型、FileSystem 服务（文本 IO + 可选版本守卫原子写）、fs/* 策略事件词汇	工具
@deepseek-ai/dsh-tool-fs	模型侧文件系统工具（read、write、edit），基于 ctx.fs	工具
@deepseek-ai/dsh-tool-fs-search	模型侧文件发现工具（glob、grep），内置打包的 ripgrep 二进制	工具
@deepseek-ai/dsh-tool-str-replace-editor	模型侧编辑工具：查看、创建、字面量替换、行插入（工具结果含 locations → 产物追踪）	工具
@deepseek-ai/dsh-shell	抽象 bash 执行器 seam（ctx.shell）	工具
@deepseek-ai/dsh-tool-bash	模型侧 bash 工具，可选通用后台任务与沙箱升级支持	工具
@deepseek-ai/dsh-web	抽象 web 访问能力 seam（ctx.web）：search/fetch 提供者注册表、与注册顺序无关的选择、请求/结果词汇、WebError 分类	工具
@deepseek-ai/dsh-tool-web	模型侧 web 工具（web_search、web_fetch），基于 ctx.web	工具
@deepseek-ai/dsh-tool-todo	模型侧待办工具 todo_write，基于事件溯源会话日志	工具
@deepseek-ai/dsh-tool-skill	模型侧技能加载工具（skill 调用入口）	工具

上下文（Context）：上下文组装与压缩

"装什么"（系统提示 + 技能目录 + 历史 + 工具 schema）与"装不下时怎么压缩"。

包名	用途	能力域
@deepseek-ai/dsh-system-prompt	系统提示组装注册表（官方插件经 systemPrompt.section() 注册提示片段）	上下文
@deepseek-ai/dsh-compaction	抽象压缩服务 seam	上下文

	（ctx.compaction）	
@deepseek-ai/dsh-compaction-basic	token 计量驱动的压缩策略 + LLM 摘要后端（检测溢出 → 修剪 → 摘要）	上下文
@deepseek-ai/dsh-skill	Agent 技能提供者注册表（技能目录注入上下文，模型按需调用）	上下文

会话（Session）：持久化、标题、遥测

包名	用途	能力域
@deepseek-ai/dsh-session	事件溯源会话存储（event-sourced session store）	会话
@deepseek-ai/dsh-session-title	基于日志的会话标题服务与提供者注册表	会话
@deepseek-ai/dsh-session-telemetry	遥测 seam：会话事件捕获、投影、脱敏、交接给上报后端	会话

子代理（Subagent）：委派子任务

包名	用途	能力域
@deepseek-ai/dsh-subagent	抽象子代理 seam（ctx.subagents）：委派子代理的命名提供者注册表	子代理

MCP：外部工具服务器接入

包名	用途	能力域
@deepseek-ai/dsh-mcp-client	MCP 客户端桥：连接 MCP 服务器并把其工具注册到 ctx.tools	MCP

工作流（Workflow）：多步确定性编排

包名	用途	能力域
@deepseek-ai/dsh-workflow	工作流能力 seam：ctx.workflows 服务、run 词汇、workflow/* 事件	工作流

安全（Safety）：沙箱、审批、循环卫生

包名	用途	能力域
`@deepseek-ai/dsh-sandbox`	抽象进程沙箱 seam（`ctx.sandbox`）：同世界隔离词汇 + SandboxProvider 契约	安全
`guard/*`	循环卫生与工具超时（白皮书 8.5 提及；npm 上具体包名待补全）	安全
`interaction/*`	权限/审批（危险操作确认；npm 上具体包名待补全）	安全

LLM：模型接入与策略

包名	用途	能力域
`@deepseek-ai/dsh-llm`	Provider 无关的 LLM 服务接口	LLM
`@deepseek-ai/dsh-llm-deepseek`	DeepSeek chat-completions 适配器（默认 `provider=deepseek-official`，接受 off/high/max 档位）	LLM
`@deepseek-ai/dsh-llm-retry`	Provider 路由的 LLM 请求重试策略	LLM

UI：浏览器界面与客户端服务

包名	用途	能力域
`@deepseek-ai/dsh-client-runtime`	客户端核心服务：SlotsService、SessionsService（作用域树 + 对象层）	UI
`@deepseek-ai/dsh-client-locale`	语言包插件：Host 端 zh/en 偏好、浏览器端回退、语言快照、类型化命名空间字典	UI
`ui-conversation` / `ui-tool` 等	Web UI 各部件（白皮书 8.1 提及 `client/*`；npm 上具体包名待补全）	UI

已知名但 npm 未发布（历史 404 包）

以下包名仅存在于白皮书踩坑记录中，**在 npm 上无法安装**（rc.1 时代依赖断裂的根源），遇到 404 属正常：

包名	说明
@deepseek-ai/dsh-pty	rc.1 时代依赖，从未发布（pnpm dlx 404 的根因，见第 2 章 FAQ）
@deepseek-ai/dsh-type-meta	rc.1 时代依赖，从未发布（rc.1 依赖断裂的根因，见第 3 章 3.5 节）

规避方式：依赖统一走 ^0.1.0-rc.6 线。

相关章节：第 8 章 · 工具与上下文系统（60+ 能力包地图）· 第 9 章 · MCP 子代理与工作流 · 术语表与命令速查

Appendix C · benchmark (Chinese original)

附录 C：同模型 × 不同 Agent 实测对比（Benchmark）

实测日期：2026-08-13 ｜ 模型：deepseek-v4-flash（同一 opencode-go 网关，同一 API key） ｜ Agent：dsh / opencode / omp

*目的：**控制模型变量，对比 Agent 工程层的差异**（提示词、工具链、轮次管理）。*

扩展：2026-08-13 追加 T4（生成 markdown 报告）/ T5（多文件小重构），样本 3 轮 → 5 任务 × 3 轮。

方法

项	说明
模型	deepseek-v4-flash（三 agent 均走 opencode-go 网关 + 同一 key，公平对照）
Agent	dsh（headless profile）、opencode（opencode run）、omp（--print 非交互）
任务	T1 创建文件 / T2 搜索统计 / T3 修复 bug + 验证 / T4 读数据生成报告 + 自复核 / T5 多文件重构 + 测试通过
环境	Windows 11 + Node 24，独立工作目录，无预置上下文
度量	耗时（秒）+ 正确性（人工核对产物/输出）

任务说明（T4/T5 为合成数据，无隐私风险）：

任务	输入	要求	人工核对标准
T4 生成报告	data.json（4 条合成销	生成 report.md：数据	汇总数字与数据一致

	售记录）	表格 + 汇总（总销量/总营收/最畅销），自复核	（397 / 130853 / 耳机）
T5 多文件重构	`mathops.py` + 依赖它的 `main.py`（含断言）	把 `multiply` 的乘法逻辑提取为 `_multiply` 辅助函数，不改 `main.py`，`python main.py` 通过	出现 `_multiply` 且 multiply 委托调用、断言全过

结果（3 轮采样，取中位数）

Agent	T1 创建文件	T2 搜索统计	T3 修 bug+验证	T4 生成报告	T5 多文件重构	总计（中位）	正确率
omp	10s ✅	11s ✅	15s ✅	21s ✅	13s ✅	**70s**	45/45
dsh	22s ✅	15s ✅	48s ✅	26s ✅	19s ✅	**130s**	45/45
opencode	35s ✅	37s ✅	42s ✅	30s ✅	28s ✅	**172s**	45/45

总计 = 5 任务中位数相加；扩展前（仅 T1-T3）为 omp 36s / dsh 85s / opencode 114s。

原始 3 轮（秒）：

Agent	轮次	T1	T2	T3	T4	T5
dsh	1 / 2 / 3	11 / 27 / 22	11 / 15 / 22	27 / 48 / 74	30 / 26 / 25	19 / 25 / 18
opencode	1 / 2 / 3	18 / 35 / 35	19 / 37 / 37	50 / 41 / 42	30 / 49 / 30	28 / 28 / 32
omp	1 / 2 / 3	9 / 10 / 12	9 / 11 / 12	13 / 33 / 15	23 / 21 / 17	16 / 12 / 13

解读

1. **正确率三家持平（45/45）**——5 任务 × 3 轮全部正确。同模型下，Agent 工程层的差异主要体现在**效率**而非**能力上限**；即便引入文档生成与多文件重构两类新任务，正确性差异仍未出现。

2. **总耗时稳定排序**：omp < dsh < opencode（3 轮一致，中位数相加 70s / 130s / 172s）。

3. **任务类型对效率差异的影响**（本次扩展的核心发现）——按耗时画像，5 个任务可归为三类：

- **单步轻量**（T1 创建文件 / T2 搜索统计）：agent 间差距**最大**。opencode 35/37s vs omp 10/11s（约 **3.4×**）。任务越简单，越接近纯"启动 + 一次工具调用"开销，opencode 的启动/框架开销占比越高。

- **多步执行 + 验证**（T3 修 bug + 跑验证 / T5 多文件重构 + 测试）：dsh 与 opencode 在此类任务上差距**显著收窄**（T5：19s vs 28s；T3：48s vs 42s 甚至反超）。多工具链场景下，双方都要多次读文件/改文件/跑命令，框架层"每步思考"成本拉平了启动开销差异；dsh 的优势在 T5 上最明显（19s，接近 omp 的 13s），而 T3 的 bug 定位仍让 dsh 偏慢。
- **文档生成**（T4 读数据 → 写报告 → 自复核）：agent 间差距**最小**（omp 21s / dsh 26s / opencode 30s，约 1.4×）。报告生成是"读一次 + 写一次长文本"的回路，耗时主要由模型**输出 token 数**主导，Agent 工程层能优化的工具往返很少，故三家趋同。

4. **反直觉点：T5（重构+测试）普遍比 T4（写报告）更快**（dsh 19s<26s、omp 13s<21s；仅 opencode 28s≈30s）。原因：重构是"短文本代码编辑 + 秒级 `python main.py` 验证"的回路，每步模型思考短；而写 markdown 报告需要一次较长文本生成，输出时间占比高——与第 6 章"工具链任务 90% 时间在模型思考/生成"的性能模型一致。

5. **omp 的领先幅度随任务类型变化**：搜索统计类最大（T2 为 opencode 的 3.4×），文档生成类最小（T4 仅 1.4×）。说明 omp 的优势主要在**轮次少、工具调用快**，而非模型本身；当任务由长文本生成主导时，任何 agent 的工程层都难再压缩模型输出时间。

6. **波动性提示**：opencode 在 T4 第 2 轮出现 49s 离群（其余两轮 30s），dsh 在 T3 波动最大（27→74s）。多轮中位数能吸收这类网络/工具链抖动，单轮比较需谨慎。

7. **dsh 定位**：dsh 居中，且 headless 是"运行时 + 插件生态"——**同样的模型，通过插件（如提速插件降推理档）可进一步优化耗时**（见第 6 章）；T5 实测已接近 omp，说明多文件重构类任务是其相对强项。

8. **样本警示**：3 轮 × 5 任务，同网关同 key，含网络抖动——方向可信，绝对值会随环境波动。

可复现

```
# 三个 agent 的命令（独立目录）
dsh --profile headless "任务"
opencode run "任务" --model opencode-go/deepseek-v4-flash
omp "任务" --model deepseek-v4-flash --print
```

T4/T5 任务原文（T1-T3 见上一版本记录）：

```
T4：读取当前目录 data.json 中的数据，生成 markdown 报告 report.md：包含每行数据的表格（产品/销量/单价三列）和汇总（总销量、总营收、最畅销产品），生成后自己复核汇总数字与 data.json 一致，完成后回复 已生成
T5：重构 mathops.py：把 multiply 函数内的乘法逻辑提取为私有辅助函数 _multiply(a, b)，multiply 改为调用 _multiply；main.py 依赖 mathops.py 并包含断言测试，不得改动 main.py；重构后运行 python main.py 验证所有断言通过，完成后回复 已重构
```

完整任务定义、合成数据与产物：~/agent-bench/bench-ext/（T4/T5 每 agent 独立目录）；T1-T3 见 ~/agent-bench/。白皮书仓库内 benchmark/ 目录规划中。