

Dashboard Improvement Plan

Repository: /home/mv/projects/developer-dashboard Focus: startup performance of the public dashboard command

Executive Summary

The public dashboard command is thin by design, but it still matters because it is the most frequently used entrypoint. Profiling shows that the switchboard itself is not the majority of end-to-end time for a normal built-in command. The bigger cost is split between Perl module loading and repeated current-directory resolution, with additional command-specific cost inside the helper process after handoff.

Key Measurements

Probe	Observed Time	Meaning
dashboard version	about 110 ms average	Special-case fast path. Useful baseline, but not representative of normal helper dispatch.
dashboard paths	about 493 ms average	Representative built-in command path through the switchboard and staged helper.
Direct helper: ~/.developer-dashboard/cli/dd/ paths	about 362 ms average	Shows most of the cost is after the public switchboard hands off.
perl -Ilib -c bin/dashboard	about 120 ms	Compile/load overhead is a major part of the switchboard cost.
Switchboard module load	about 77 ms	Static startup cost from Perl module loading before command work begins.
Isolated env/path/helper setup	about 23 ms total	PathRegistry, EnvLoader, and ensure_helpers are not the main cost in isolation.

Dispatch Shape Benchmarks

I also measured several real dispatch shapes to compare the public dashboard path against

the direct resolved target for the same command. These measurements were run as 10-command loops.

Shape	Command	dashboard average	direct average	extra routing overhead
built-in helper	dashboard ps1 --jobs 0 --cwd /home/mv/ projects/ developer- dashboard	about 870 ms	about 801 ms	about 69 ms
top-level CLI script	dashboard system-temp cpu	about 177 ms	about 17 ms	about 160 ms
skill CLI	dashboard ssh.list	about 1578 ms	about 104 ms	about 1474 ms
nested skill CLI	dashboard sk.system -status.disk /	about 1515 ms	about 60 ms	about 1455 ms

Dispatch Interpretation

- * ps1 is not dominated by the public switchboard. Most of its time is in the command body itself.
- * Top-level custom CLI scripts are much more sensitive to public dispatch overhead than built-in helpers.
- * Skill CLI and nested skill CLI dispatch are by far the heaviest routing shapes measured so far.
- * The dotted skill dispatch path is the next clear profiling target after the public startup path.

Problem

The command is thin in architecture, but the real startup path still pays a measurable tax before useful work happens. This matters because users hit dashboard for nearly everything, so even modest startup delays become visible and repetitive friction.

Primary Problem

Normal built-in command dispatch is slower than expected because the runtime repeatedly resolves the current working directory and project context, and that cost compounds across both the public switchboard and the helper process.

Secondary Problem

The command-specific helper runtime is doing enough repeated path recomputation that even a lightweight command such as dashboard paths still feels expensive.

Tertiary Problem

Dotted skill command dispatch, especially nested skill dispatch, carries a very large routing cost compared with the direct command body. That makes skill-backed command shapes disproportionately expensive even when the underlying script is cheap.

Evidence

1. A strace of dashboard paths showed `/bin/pwd` executing 72 times during one command invocation. 2. Split of those `pwd` executions: - 6 before helper handoff - 66 after helper handoff 3. A direct probe confirmed that `Cwd::cwd()` on this machine forks `/bin/pwd`. 4. A direct microbenchmark showed `cwd()` averaging about 2.0 ms per call on this machine. 5. The paths helper alone took about 212 ms when invoked directly through its Perl module path. 6. Dispatch-shape benchmarks showed: - built-in helper dispatch has a moderate handoff cost - top-level custom CLI dispatch has a larger fixed routing cost - skill and nested skill dotted dispatch have very large routing overhead relative to direct execution

Root Cause

1. Repeated `cwd()` Calls

The largest concrete hotspot is repeated `Cwd::cwd()` use. On this environment, that is not a cheap in-process call. It forks `/bin/pwd`. That makes otherwise innocent path lookups surprisingly expensive.

2. Repeated Project and Runtime Path Derivation

`PathRegistry` methods derive current project roots, runtime layers, state roots, and related values repeatedly. Commands such as dashboard paths ask for many of these values in one call, and that compounds the `cwd()` overhead.

3. Startup Compile and Load Cost

The switchboard still pays a meaningful Perl compile/load cost before it can decide where to hand off. That cost is not the majority of total runtime for a normal helper path, but it is a large

chunk of the public endpoint overhead.

4. Missed Caching Opportunity

Developer::Dashboard::CLI::Paths::_build_paths() passes `cwd => cwd()` into `PathRegistry->new()`, but `PathRegistry` does not appear to store or reuse that value. So the helper pays for `cwd()` but does not gain any caching from it.

5. Expensive Dotted Skill Dispatch

The skill and nested-skill command path is materially heavier than built-in helper and top-level CLI dispatch. The benchmark gap is too large to explain by command-body work alone. That points to runtime setup, command resolution, environment layering, and nested-skill discovery as major cost centers in the dotted dispatch path.

What Is Not the Main Problem

- * `EnvLoader` by itself is not the primary cost center.
- * `InternalCLI::ensure_helpers()` is not the primary cost center in isolation.
- * The thin switchboard architecture itself is not the problem. The issue is the repeated runtime work done around path discovery and module startup.

Recommendation

Keep the public dashboard command thin. Do not replace the switchboard design. Instead, reduce repeated startup work, especially current-directory and project-root recomputation.

Recommended Priority Order

1. Eliminate repeated `cwd()` calls from the startup and helper path. 2. Add request-local caching inside `PathRegistry` for `cwd`, project root, runtime layers, and derived roots. 3. Profile and optimize dotted skill dispatch, especially nested skill command routing. 4. Audit helper commands like `paths` for repeated calls into path-derivation methods that can be computed once. 5. Only after that, consider deeper module-load reduction in `bin/dashboard`.

Solution Plan

Phase 1: Remove High-Volume `cwd()` Cost

- * Store a single `cwd` value per invocation and reuse it throughout the command lifetime.

- * Teach PathRegistry to accept and actually use a precomputed cwd.
- * Replace repeated internal cwd() lookups with the cached value wherever the command is operating in one fixed current directory.

Phase 2: Add Request-Local PathRegistry Memoization

- * Memoize:
 - * current project root - runtime layers - runtime root - state root and derived roots - CLI root and similar derived directories
 - * Keep memoization strictly per-process and per-invocation so it does not create stale cross-command state.

Phase 3: Reduce Repeated Helper-Side Computation

- * For helper commands like paths, compute the path inventory once and reuse that data structure rather than calling multiple lazy getters that each recompute upstream state.
- * Review all_paths() and related accessors for repeated tree walks and repeated state-root creation.

Phase 4: Module Load Cleanup

- * Move non-essential use statements behind require when the command path does not always need them.
- * Keep the switchboard's eager module set as small as possible.
- * Preserve readability and safety; do not trade away clear error behavior for micro-optimizations.

Phase 5: Dotted Skill Dispatch Optimization

- * Profile the ssh.list and sk.system-status.disk / paths specifically.
- * Measure where time goes inside:
 - * skill name resolution - nested skill discovery - env-layer loading - command path selection - repeated cwd and project-root derivation
- * Add request-local caching for nested skill lookup results where safe.
- * Avoid recomputing the same nested skill chain multiple times during one invocation.

Expected Outcome

The biggest likely gain is from removing repeated /bin/pwd subprocess calls. Since the trace showed 72 pwd executions for one dashboard paths command, reducing that to one or a small handful should materially reduce wall time. After that, request-local caching should cut the remaining repeated path derivation overhead.

The second biggest likely gain is from optimizing dotted skill dispatch. The measured gap between dashboard ssh.list and the direct skill CLI, and between dashboard sk.system-status.disk / and the direct nested skill CLI, is large enough that improving command routing there should produce obvious user-facing wins.

Implementation Notes

- * Any optimization work should preserve the DD-OOP-LAYERS behavior and lookup order.
- * Do not hide errors or weaken explicit runtime checks.
- * Re-profile after each phase instead of batching all changes blindly.

Suggested Verification After Changes

1. Re-run timing probes for dashboard version and dashboard paths. 2. Re-run syscall trace and count /bin/pwd executions. 3. Verify helper behavior and path outputs remain identical. 4. Re-run the dispatch-shape benchmarks for: - dashboard ps1 - dashboard system-temp cpu - dashboard ssh.list - dashboard sk.system-status.disk / 5. Run the relevant CLI, path, and skill-dispatch regression tests.

Conclusion

The public dashboard command is not slow because the switchboard pattern is wrong. It is slow because repeated path discovery work, especially cwd() on this environment, is expensive and occurs many times per invocation. On top of that, dotted skill dispatch is carrying a very large routing tax. The strongest plan is to keep the architecture, remove repeated cwd and process calls, add request-local path caching, and then optimize the dotted skill routing path before trimming deeper module startup costs.