LispM: A Live-Image AI Agent Runtime with a Lock-Free CSP Sidecar Kernel

Ryan Kung
LispMachineLab
ryankungs@gmail.com

Preprint — November 2026

Abstract

Modern AI coding and desktop agent harnesses execute outside the user's interactive software environment, communicating with editors and applications via file-system side effects, sprawling Model Context Protocol (MCP) schemas, or brittle pixel-scraping interfaces. Inverting this architecture—hosting agents directly as first-class objects inside a self-documenting *live runtime image* (in the Lisp Machine and Smalltalk tradition, realized atop GNU Emacs)—unifies the editor's unsaved buffers, undo trees, and introspection primitives with the agent runtime, collapsing dozens of ad-hoc tool wrappers into live image evaluation. However, hosting concurrent agents inside a 40-year-old single-threaded, stop-the-world garbage-collected Lisp image exposes a severe responsiveness ceiling: scanning multi-megabyte agent transcripts, searching repositories, negotiating TLS, or spawning subprocess bursts stalls the host event loop for hundreds of milliseconds.

We present LispM, an in-image agent runtime that resolves this tension through a *dual-Lisp split-kernel architecture*. Stateful workspace semantics—editable Org-mode context buffers, dirty-buffer coherence, policy-mediated Lisp actions (`run_lisp`), out-of-tree Git turn journals, and condition/restart recovery—remain inside GNU Emacs's single-threaded image. Latency-critical parallel workloads are offloaded over a zero-eval S-expression wire protocol to `lispm-kernel`, a crash-isolated Steel Bank Common Lisp (SBCL) sidecar process coordinated exclusively via *lock-free Communicating Sequential Processes (CSP)* without a single domain mutex. Inside `lispm-kernel`, a single-owner sequential scheduler, broadcast cancellation gates, a 4 MiB byte-credit backpressure sink with wait-free control bypass, and an async-signal-safe CAS wakeup pipe drive parallel worker pools while Emacs ingests results in 3 ms frame-clock slices that yield immediately to keystrokes. Across 12-run benchmarks over 256 session transcripts (68.6 MB) alongside eight active streaming terminals, LispM cuts median worst-case UI stall from 196.31 ms in pure Emacs Lisp (and 6.64 ms in an earlier mutex sidecar) to **0.54 ms** (**366× lower**; **0.25 ms** max stall across a real 314.7 MiB developer trace, **1,268× lower**) with zero host garbage collections, while `run_lisp` replay continuations reduce multi-file refactoring prompt payloads by **178×** and prefix-folded Org parsing scales to 120k-token contexts in **3.49 ms**.

Keywords: AI Agent Runtime, Live Lisp Image, Communicating Sequential Processes (CSP), Lock-Free Concurrency, Split-Kernel Architecture, GNU Emacs, Common Lisp

1 Introduction

The prevailing architecture for AI coding and computer-use agents treats the user's interactive environment as an external, opaque target. Standalone CLI harnesses (such as Claude Code, Codex, and SWE-agent [24]) maintain their own conversation loops, permission prompts, and file readers in separate terminal processes, mutating source files on disk behind the editor's back. General-purpose computer-use harnesses go further, capturing screenshots and synthesizing mouse clicks because they share neither memory nor object representations with the applications they drive.

This *external-harness architecture* imposes three structural costs. First, **state divergence**: external tools read committed disk bytes rather than live editor buffers, silently clobbering or ignoring unsaved human edits and bypassing buffer-level undo histories. Second, **schema bloat with bounded expressiveness**: every new capability—mail, browser tabs, version control, diagnostics, symbol cross-references—requires another JSON tool schema injected into the prompt [1], consuming tens of thousands of context tokens while leaving anything outside the predefined tool menu unreachable. Third, **opaque context and control**: conversation history is trapped in an append-only log that users cannot directly edit, annotate, or prune in place, and heterogeneous CLI agents running side by side cannot inspect or delegate to one another.

Half a century ago, MIT CONS/CADR [4] and Symbolics Genera [15, 21] demonstrated an alternative system organization: the *Lisp Machine*, where the editor, shell, debugger, window system, and user programs coexisted as live, inspectable Lisp objects inside a single persistent image. Today, GNU Emacs [14, 20] preserves this self-documenting image architecture across modern operating systems. Every

buffer, window, directory tree, compiler diagnostic, and interactive command is a documented Lisp function carrying online metadata, arity, and source locations.

Inverting the agent architecture so that *Emacs itself is the agent harness* immediately eliminates the external-harness tax. In our runtime, LispM, an agent's conversation transcript *is* an editable Org-mode buffer that parses byte-for-byte into the model's context; tool calls act directly on live buffers as atomic undo steps; human approvals are Org `TODO` state transitions; failed turns pause at resumable Lisp Machine *conditions* with interactive *restarts* [18]; and a compact core toolset combined with policy-mediated Lisp programs (`run_lisp`) and live image reflection (`eval_elisp`) replaces sprawling wrapper catalogs.

Yet running modern multi-agent workloads inside GNU Emacs exposes a fundamental systems bottleneck: **the single-threaded image ceiling**. Like CPython prior to free-threading [5] and classic Smalltalk [9], Emacs executes Lisp bytecode, native-compiled Elisp, process filters, redisplay, and mark-and-sweep garbage collection on a single OS thread [14]. Rewriting Emacs's C core for preemptive multi-threading would break four decades of extensions that mutate global buffer and dynamic-binding state without locks. When multiple coding agents operate concurrently—indexing hundreds of megabytes of JSONL session histories, searching thousands of project files, negotiating TLS handshakes, or spawning bursts of external Git and shell commands—Emacs's main thread stalls for 50–500 ms at a time, dropping display frames and freezing keystroke echo.

Embedding native C or Rust worker threads into Emacs via dynamic FFI modules only partially mitigates this problem and introduces two severe hazards: (1) a single memory or FFI fault inside the shared address space crashes the entire editor along with every unsaved buffer, and (2) marshaling complex Lisp structures across C FFI boundaries incurs main-thread allocation spikes. Furthermore, our early multi-threaded sidecar prototype revealed that conventional mutex-protected task queues, caches, and subprocess monitors still suffer from tail-latency spikes under contention, broadcast thundering herds during task cancellation, and signal-safety hazards around `SIGCHLD`.

To resolve this tension without sacrificing either live-image unity or sub-millisecond interactivity, we design and implement a **dual-Lisp split-kernel architecture** (Figure 1). LispM partitions responsibilities along a strict state-ownership boundary: all mutable workspace state, live buffers, user interaction, and security policies remain in GNU Emacs, while compute- and I/O-intensive operations execute in `lispm-kernel`, an isolated multi-threaded Common Lisp sidecar compiled to a standalone native binary via SBCL [19]. The two Lisp runtimes communicate over standard streams using a non-evaluating, line-oriented S-expression wire protocol. Internally, `lispm-kernel` is structured from the ground up around Hoare's Communicating Sequential Processes (CSP) [7, 8] with *zero domain mutexes*, paired with frame-clock time-sliced message ingestion on Emacs's side.

This paper makes the following contributions:

- **Live-Image Agent Runtime (§3):** We present the design of LispM (`lispm-agent`), formulating an in-image harness where editable Org buffers serve as prefix-cache-aware LLM contexts, `run_lisp` provides static/dynamic purity-checked code actions with replay-based continuation across human approvals, and monotonic permission ceilings govern heterogeneous multi-agent delegation.
- **Lock-Free CSP Sidecar Kernel (§4):** We architect `lispm-kernel` around four lock-free CSP primitives: single-producer/multi-producer mailboxes, one-shot broadcast `cancel-token` gates, a 4 MiB byte-budgeted `credit-sink` with zero-wait control bypass, and a work-helping `work-group` pool, coordinated by a single-owner sequential scheduler ($P_{\text{scheduler}}$) and an async-signal-safe 64-bit CAS `wakeup-pipe` for `poll(2)` subprocess multiplexing.
- **Coherent Split-Lisp Subsystems (§5):** We show how to preserve live-buffer truth across process boundaries in incremental nanosecond-stamped session indexing (P_{cache} immutable snapshots), hybrid parallel codebase search (`search-files`, which delegates dirty open buffers back to Emacs), and non-blocking TLS/subprocess execution.
- **Comprehensive Kernel & Harness Evaluation (§6):** Across 12-repeat synthetic suites and a real 314.7 MiB developer trace (single transcripts up to 108 MiB), LispM cuts median max UI stall from 196.31 ms in Elisp and 6.64 ms in a mutex sidecar to **0.54 ms** (**366**× and **12.4**× lower) under 8-terminal load and **0.25 ms** (**1,268**× lower) on the real trace; sustains **150.0 kreq/s** CSP dispatch and **0.58 ms** `poll(2)` cancellation; cuts 40-file refactoring prompt tokens by **178**× via `run_lisp` replay continuations (~3.7 ms/pass); and parses 120k-token Org contexts incrementally in **3.49 ms**.

2 Background and Motivation

2.1 From Lisp Machines to Modern Agent Harnesses

On classic Lisp Machines [4, 15, 21], there was no boundary between the interactive shell, the editor (Zmacs), and system utilities: every artifact on screen was a typed *presentation* backed by a live Lisp object, errors entered a resumable condition system rather than terminating processes, and extending the system meant defining a Lisp function in the running world.

Modern LLM agent frameworks [12, 17, 24, 25] attempt to reconstruct an operating environment for autonomous agents from the outside in. Because an external harness runs in an isolated Python, Node.js, or Rust CLI process, it cannot directly read an editor's modified in-memory buffers, inspect active compiler diagnostics, or push reversible edits onto the user's undo ring. Bridging this gap via Model Context Protocol (MCP) [1] servers helps expose discrete endpoints, yet

Table 1: Architectural comparison of AI agent harnesses and interactive host runtime designs.

Architecture	Live Unsaved Buffer Truth	Context Schema Cost	Crash Isolation	True Multi-Core Parallelism	Lock/GC Stall Freedom on UI	Targeted Sub-ms Cancellation
External CLI Harness (Claude Code, Codex, SWE-agent [24])	No (disk only)	High (N wrappers)	Yes (OS proc)	Partial	Yes (external)	No (PTY/poll)
GUI Computer-Use Harness (pixel + click agents)	No (pixels)	Medium–High	Yes (OS proc)	Partial	Yes (external)	No
Pure Single-Thread Image (Elisp-only agent packages)	**Yes**	**Low** (reflection)	N/A	No (GIL / 1 thread)	No (50–500 ms stalls)	No
In-Process Native FFI Module (C/Rust `emacs-module`)	**Yes**	**Low**	**No** (segfault kills host)	Yes	Partial (FFI alloc/GC)	Partial
Mutex Sidecar Kernel (`lispm-kernel` v0.8)	**Yes**	**Low**	**Yes**	Yes	Mostly (5–7 ms spikes)	Contended condvars
LispM Dual-Lisp CSP Split-Kernel (v0.9, Ours)	**Yes**	**Low**	**Yes**	**Yes** (SBCL pools)	**Yes (<1 ms max)**	**Yes (CAS gates/pipes)**

rapidly inflates prompt prefixes—a single GUI automation MCP server of 26 tools consumes roughly 14,000 tokens on every request—while confining the model to a static menu of operations. CodeAct [22] improves compositionality by letting agents emit Python or Bash scripts, but executing arbitrary scripts outside a fine-grained host permission model forfeits per-operation human approval and buffer-level atomicity.

Table 1 contrasts existing points in the design space against LispM. Hosting agents directly inside GNU Emacs recovers the Lisp Machine's unified object world, but immediately confronts the physical limits of Emacs's execution engine.

2.2 Anatomy of Stalls in Single-Threaded Image Hosts

GNU Emacs's C runtime executes Lisp evaluation, timer callbacks, subprocess I/O filters, and display layout on one thread [14]; even cooperative Lisp threads introduced in Emacs 26 multiplex onto that single lock and cannot execute in parallel across cores. Under multi-agent workloads, we observed four distinct sources of interactive freezes (quantified in §6):

1. **Transcript & Index Scanning + GC Amplification:** Listing and resumability across LispM, Claude Code, and Codex sessions require scanning hundreds of JSONL transcripts on disk (tens to hundreds of megabytes) to extract session titles, working directories, and turn-completion states. Parsing JSON lines in Elisp allocates millions of short-lived strings and cons cells, triggering 4–14 stop-the-world garbage collections per scan and stalling keystrokes for 200 ms.

2. **Subprocess Creation Storms:** Even when using Emacs's asynchronous `make-process` rather than synchronous `call-process`, spawning a burst of 32 short-lived Git or shell helpers stalls the main thread for over 53 ms due to synchronous descriptor setup, environment array marshaling, `posix_spawn`/`fork` overhead, and sentinel teardown.

3. **Main-Thread TLS & Network Stalls:** Emacs's built-in `url.el` negotiates TLS handshakes on the main thread; refreshing package archives or fetching artifacts through an HTTP/SOCKS proxy freezes the cursor for hundreds of milliseconds per connection.

4. **Repository-Wide Search:** Searching thousands of project files synchronously in Elisp blocks input for 70–1,300 ms; time-slicing Elisp into 30 ms chunks avoids complete freezes but stretches wall-clock latency by 6× compared to multi-core native search.

2.3 Evolution Beyond FFI Modules and Mutexes

Our initial attempt to offload heavy work used a Rust dynamic module (`emacs-module.h`) loaded into Emacs's address space. While Rust remains well-suited for isolated leaf components with narrow interfaces—such as LispM Desktop's off-thread Alacritty terminal emulator—using in-process FFI for a general-purpose agent kernel proved brittle: any panic, C library fault, or FFI lifetime bug terminates the host Emacs process and destroys unsaved buffers, while translating deeply nested S-expressions across C pointers adds complexity and main-thread allocation.

Moving the kernel into an out-of-process SBCL Common Lisp executable (`lispm-kernel` v0.8) gave us process crash isolation, native multi-core threads without a GIL, isolated garbage collection, and native S-expression symmetry. However, v0.8 relied on conventional mutexes and condition variables around shared priority queues, output buffers, and transcript caches. Under heavy load (e.g., eight streaming agent terminals competing with cold session scans), mutexes introduced three pathologies: (1) fast admission and status paths queued behind contended locks, producing 5.4–6.6 ms UI stall spikes; (2) cancelling a single superseded job required condition-variable broadcasts that woke unrelated workers only to re-contend; and (3) composing locks with asynchronous POSIX `SIGCHLD` process-exit hooks risked deadlocks or forced fallback to 10–50 ms sleep-polling loops.

Why not adopt the Actor model [6]? Unbounded actor mailboxes obscure overload: a fast producer scanning directories or streaming command output floods memory before Emacs can drain it, and splitting sequential I/O lifecycles across asynchronous actor turns complicates structured `unwind-protect` resource cleanup. By contrast, Hoare's CSP [7, 8] connects a small, static topology of sequential loops via explicit channels with bounded credit backpressure and broadcast cancellation gates, matching systems kernel requirements cleanly.

3 Image-Native Agent Harness

Before detailing the sidecar kernel, we formalize how LispM (`lispm-agent`) embeds autonomous agents inside the live Emacs image. Every traditional external-harness subsystem maps onto a first-class Emacs abstraction.

3.1 Agent Object Model and Permission Ceilings

LispM separates static capability specifications (*roles*) from live project conversations (*agents*) and persistent cross-project identities (*characters*):

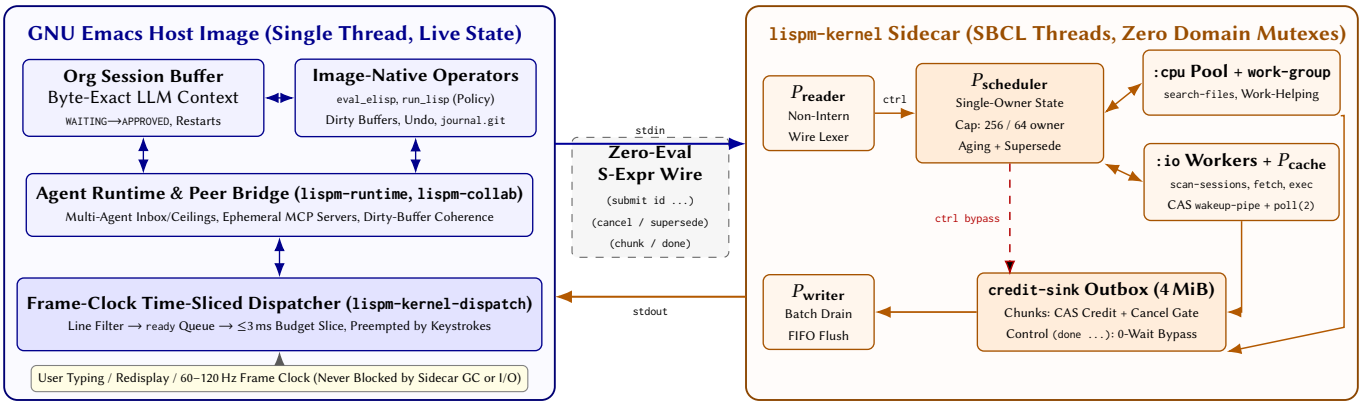


Figure 1: Architecture of LispM. Stateful editing, live buffer coherence, permission gating, and human-in-the-loop restarts reside inside GNU Emacs's single-threaded Lisp image (left). Latency-critical parallel I/O, transcript indexing, regex search, and subprocess multiplexing run in an isolated multi-threaded SBCL sidecar, `lispm-kernel` (right), coordinated purely through lock-free CSP channels without domain mutexes.

- A **Role** (ρ) defines instructions, an allowed tool set $\mathcal{T}_\rho$, a permission policy $\pi_\rho : C \to \{\texttt{auto}, \texttt{ask}, \texttt{deny}\}$ over nine capability classes (C: `read`, `memory`, `comm`, `lisp`, `write`, `shell`, `exec`, `web`, and `mcp`), and model parameters.
- An **Agent** (a) is an active session of role ρ rooted at a directory (local path or remote TRAMP target such as `/ssh:vm:/work/`). Crucially, an agent is an in-memory Lisp struct rather than an OS thread or process, transitioning through a five-state event-loop machine—`idle` → `requesting` → (`acting` | `waiting`) → `idle`, plus `compacting` during context handover. Ten concurrent agents awaiting model responses or human approvals consume near-zero host memory and zero OS threads.

Monotonic Delegation and Cycle-Free Wait Graphs. Agents collaborate asynchronously via `send_message` and `ask_agent`. When caller a_1 delegates a subtask to a subagent a_2 of role ρ_2, LispM enforces a strict monotonic permission ceiling under the total order `deny` $\prec$ `ask` $\prec$ `auto`:

$$\forall c \in C, \quad \pi_{a_2}(c) = \min_{\prec}\big(\pi_{\rho_2}(c),\ \pi_{a_1}(c)\big) \tag{1}$$

This prevents confused-deputy privilege escalation: a read-only `reviewer` role ($\pi(\texttt{write}) = \texttt{deny}$) cannot achieve file modifications by spawning a `coder` subagent. Because an agent may issue parallel `ask_agent` calls in one response, active waits form a directed graph $G = (V, E)$; before admitting a delegation ($a_1 \to a_2$), the runtime traverses G to reject any edge that closes a dependency cycle. Both the caller's `:ASKED:` call property, the callee's ceiling (`#+lispm_ceiling:`), and pending `inbox.org` entries are persisted to disk so interrupted multi-agent delegations resume seamlessly across editor restarts.

Zero-Modification Heterogeneous Agent Bridge. Through `lispm-agents-bridge`, external CLI and daemon agents running inside Emacs terminals (Claude Code, Codex, Hermes, Holon, OpenClaw) participate as peers in the same collaboration graph without modifying their binaries. Each terminal receives an ephemeral localhost MCP server authenticated by a per-session token (exposing seven collaboration and skill tools plus read-only views of Emacs selections, dirty buffers, and Flymake diagnostics), while incoming peer messages are injected into the CLI's input composer *only* when `lispm-kernel`'s transcript scanner confirms the target's `:turn-ended` predicate is true—preventing injected newlines from accidentally answering an interactive approval prompt.

3.2 The Buffer Is the Context: Prefix-Cache Co-Design

A central architectural invariant of LispM is that *the session's Org-mode buffer is the single source of truth for the LLM context.* Prior to every API request, `lispm-transcript-parse` compiles the Org outline into the canonical Messages representation; the interactive chat UI (`lispm-chat`) is a stateless view projected from this buffer. Users can delete an Org subtree to make the agent forget a dead-end exploration, tag a heading `:ignore:` to exclude it from prompts, or add private `* Notes` headings that are never transmitted.

Treating an editable text buffer as live LLM context creates a subtle systems tension with modern model serving: frontier APIs bind preserved reasoning blocks (`thinking`) and KV prompt caches [11] to a *byte-identical conversation prefix.* LispM resolves this tension via three mechanisms:

1. **Byte-Exact Round-Tripping:** Assistant text blocks use Org source-block comma escaping for leading `*` or `#+` tokens with deterministic trailing-newline stripping, while opaque `thinking` blocks are stored verbatim as folded JSON drawers, verified by property round-trip tests.
2. **Session-Frozen Prefix Schemas:** The system prompt, initial memory index snapshot, and core tool schemas are frozen at session start. Deferred MCP/Lisp tools discovered mid-session via fuzzy `search_tools` append

monotonically to the session's tool list, incurring at most one prefix invalidation per discovery batch.

3. **Graceful Prefix Invalidation & Incremental Prefix Folding:** When a user intentionally edits earlier conversation history, requests carry `prefix_mismatch_behavior: "drop_block"`, instructing the provider to discard invalidated reasoning blocks after the edit point and rebuild the KV cache from the modified prefix forward rather than rejecting the turn. Locally, `lispm-transcript-parse` tracks the earliest buffer modification via `after-change-functions` and caches parsed prefix sections (`lispm-transcript--folded`), so normal turn appends parse only the newly appended tail in $O(|\Delta|)$ rather than re-scanning 100k+ token histories on every step.

3.3 Policy-Mediated Lisp Actions and Replay Continuations

To eliminate the granularity-versus-safety dilemma between single-step JSON tool calls and unrestricted `eval_elisp`, LispM introduces **`run_lisp`** (`lispm-lisp.el`): an effect-mediated Lisp execution substrate where agent tools appear as keyword-argument Lisp functions:

```
(dolist (file (directory-files "lisp" t "\\.el\\'"))
  (when (string-search "2025" (read_file :path file))
    (edit_file :path file :old-text "2025" :new-text "2026")))
```

Forty file inspections and selective edits collapse into a single LLM round-trip, yet every tool invocation inside the loop traverses the agent's permission policy exactly as a top-level call would (`read_file` runs automatically; `edit_file` pauses for diff approval or raises a catchable `lispm-call-denied` condition).

Because Emacs Lisp lacks delimited continuations or coroutines to suspend an active stack frame while awaiting asynchronous human approval, `run_lisp` enforces *pure prefix computation* and implements **deterministic replay continuation**:

- **Static & Dynamic Purity Confinement:** Before evaluation, an AST verifier restricts expressions to special forms, hygienic standard macros (scanning macro bodies prior to expansion to block compile-time escapes such as `eval-when-compile`), and functions marked `side-effect-free` or `pure`. At runtime, higher-order calls (`funcall`, `mapcar`) check callable allowlists, destructive list operations (`nconc`, `sort`) copy inputs, point motion outside temporary buffers requires `save-excursion`, and filename handlers (TRAMP) and text display properties are stripped.
- **Memoized Replay:** When tool call k inside a `run_lisp` program requires approval or async I/O, evaluation aborts cleanly and records call k as a `*** Asks:` child heading under the `run_lisp` Org entry. Once settled, the program re-executes from the start; calls $1 \ldots k-1$ return their memoized results in $O(1)$ without re-executing effects, deterministically reconstructing the stack state at call k.

3.4 Live Buffer Coherence, Turn Journals, and Restarts

Tools operate on live Emacs buffers rather than raw files: `read_file` observes unsaved human edits, and `edit_file` applies exact string replacements as a single Emacs undo step (`C-/`), streaming strike-through/insertion preview overlays directly into the target buffer as tokens arrive (`lispm-edits.el`). Before any tool touches a file, `lispm-tools--state` reconciles buffer and disk timestamps without interactive prompts: clean stale buffers auto-revert, whereas a buffer with unsaved human edits whose underlying file also changed on disk raises an explicit conflict error to the model rather than overwriting either side. To bound unrestricted tools (`run_shell`, `eval_elisp`), LispM snapshots the project worktree at turn boundaries into an out-of-tree Git repository (`projects/<proj>/journal.git`), enabling full-turn diffing and rollback (`lispm-revert-turn`) without polluting the user's Git history. Finally, early turn stops (network failures, step limits, context exhaustion, model refusals) signal structured Org `* WAITING Condition:` headings offering interactive Lisp Machine restarts (`retry`, `use-model`, `retry-larger`, `compact`, `abort`), governed automatically for unattended background agents via `lispm-unattended-condition-handlers`.

4 Lock-Free CSP Sidecar Kernel

To keep GNU Emacs's single thread permanently responsive while multiple agents execute heavy background workloads, LispM pairs Emacs with `lispm-kernel`, a standalone native executable compiled from Common Lisp via SBCL's `save-lisp-and-die` [19]. This section presents its wire protocol, zero-mutex CSP runtime, async-signal-safe subprocess multiplexer, and frame-clock host dispatcher.

4.1 Zero-Eval S-Expression Wire Protocol

Emacs and `lispm-kernel` communicate over anonymous POSIX pipes (`stdin`/`stdout`), framing one S-expression datum per newline-terminated UTF-8 line:

```
; Emacs -> Kernel
(submit 7 scan-sessions (:agent codex :root "~/.codex/sessions/")
        :owner 1 :generation 3 :priority 0)
(cancel 7)          (supersede 1 4)          (stats)
; Kernel -> Emacs
(hello :protocol 1 :pid 4812 :lisp "SBCL 2.6.9" :jobs (...))
(chunk 7 ((:id "..." :cwd "..." :title "..." :turn-ended t)))
(done 7 success (:files 256 :rows 256 :cached 0 :turn-ended t))
```

Although both endpoints are Lisps, using Common Lisp's standard `cl:read` on untrusted or complex editor strings would introduce severe security and symbol-table hazards: reader macros (`#.`) execute arbitrary code at read time, package-qualified symbols trigger reader errors or memory leaks via unbounded symbol interning, and Emacs Lisp's string escape syntax differs subtly from ANSI Common Lisp's (e.g., octal byte escapes vs. Unicode escapes and number-like symbols such as `1+`).

We therefore implement a dedicated, non-evaluating wire

lexer and printer (`src/wire.lisp`) paired with a structural validator in Emacs (`lispm-kernel--wire-p`):

1. **Strict Data Algebra:** Only integers, UTF-8 strings, plain symbols/keywords, proper lists, and vectors nested to depth $\leq$ 64 may cross the wire. Emacs strips text properties before `prin1` serialization and rejects floats, improper lists, or ambiguous numeric symbols.

2. **Zero Unbounded Interning:** The kernel's reader rejects quote/backquote/sharp reader macros, decodes Emacs control and hex escapes (mapping invalid surrogates or raw bytes > 127 to `U+FFFD`), and interns symbols *exclusively into the* `:keyword` *package* (length $\leq$ 256), mapping `nil` and `t` to booleans. Conversely, the kernel prints keywords with a leading colon and internal status symbols (`done`, `success`) bare lowercase so Emacs's native C `read` parses responses at native speed.

4.2 CSP Process Topology and Primitives

Inside `lispm-kernel` v0.9, *no two threads ever acquire a domain mutex or mutate shared scheduling tables.* Concurrency is expressed through four lock-free CSP primitives (`src/csp.lisp`) built directly atop SBCL's atomic compare-and-swap (`sb-ext:compare-and-swap`) and Michael–Scott lock-free queues [13] (`:sb-concurrency`):

1. **`mbox` (Fan-In/Private Channels):** Lock-free multi-producer single-consumer/multi-consumer mailboxes supporting non-blocking `mbox-send`, blocking/timed `mbox-recv`, and batch `mbox-drain`.

2. **`cancel-token` (Broadcast Cancellation Gate):** A one-shot broadcast primitive pairing an `sb-concurrency:gate` with an atomic CAS hook list. Invoking `cancel!` first opens the gate (so any thread waking from `await-cancel` immediately observes `(cancelled-p token)` = `t`) and then atomically transitions the hook list to `:cancelled`, executing registered cleanup callbacks exactly once. Registering a hook on an already-cancelled token invokes the callback inline immediately, eliminating check-then-act races.

3. **`credit-sink` (Byte-Budgeted Outbox with Control Bypass):** Standard bounded channels suffer a critical flaw in RPC multiplexing: if streaming `chunk` messages fill the channel, final `done` settlements or `protocol-error` replies also block, causing worker starvation or deadlock. `credit-sink` decouples data flow control from control progress. Data chunks atomically reserve $|s|$ bytes via CAS against a $B_{\max} = 4\,\text{MiB}$ ceiling (permitting a single oversized chunk when current occupancy is zero). When $|s|$ + waiting $> B_{\max}$, the emitting worker enqueues a local gate onto `room-waiters`, registers a `cancel-token` wakeup hook, re-verifies occupancy and cancellation after enqueueing (preventing lost wakeups), and suspends. Crucially, `credit-sink-put-control` bypasses credit reservation entirely, guaranteeing $O(1)$ wait-free enqueueing for `done`, `pong`, and `stats` frames. When P_{writer} drains a batch via `credit-sink-take`, it atomically decrements `waiting` by the summed chunk bytes and opens queued waiter gates.

4. **`work-group` (Persistent Work-Helping Subworker Pool):** To parallelize intra-job batches (e.g., scanning 256 transcripts or searching 200 files) without spawning transient OS threads per request, `work-group-map` posts a shared `batch` descriptor (holding atomic `next-index` and `in-flight` counters plus a completion gate) to $k-1$ persistent subworkers and immediately joins execution on the calling worker thread (*caller work-helping*). The final participant to decrement `in-flight` upon exhaustion or `cancel-token` firing opens the batch gate.

4.3 Single-Owner Coordinator ($P_{\text{scheduler}}$)

All request lifecycle state—active request map, per-owner request counts, owner generation numbers, and class priority queues (`:cpu` and `:io`)—is owned exclusively by a single sequential process, $P_{\text{scheduler}}$, consuming events from `control-mbox`:

$$\begin{aligned}\mathcal{E} ::= {} & \texttt{:submit} \mid \texttt{:cancel} \mid \texttt{:supersede} \\ & \mid \texttt{:worker-ready} \mid \texttt{:settled} \mid \texttt{:shutdown}\end{aligned} \tag{2}$$

Bounded Admission and Generation Superseding. $P_{\text{scheduler}}$ enforces a global cap of $N_{\max} = 256$ in-flight requests and $N_{\text{owner}} = 64$ per subsystem owner (rejecting excess submissions immediately with `(done id rejected capacity)` so callers fall back gracefully). Each UI view or subsystem tags submissions with an `:owner` ID and monotonically increasing `:generation` g. Submitting with $g > g_{\text{cur}}$(owner), or sending `(supersede owner` g`)`, advances g_{cur}; $P_{\text{scheduler}}$ immediately removes older queued requests of that owner (settling them as `cancelled`) and fires `cancel!` on running ones, while rejecting late arrivals with $g < g_{\text{cur}}$ as `stale`.

Starvation-Free Aging Dispatch. Idle workers of class $c \in \{\texttt{:cpu}, \texttt{:io}\}$ send `(:worker-ready` c m_w`)` containing their private mailbox m_w to `control-mbox`. $P_{\text{scheduler}}$ matches idle workers against the highest effective-priority request in Q_c, where priority ages linearly with wait time to prevent low-priority background scans from starving behind interactive bursts:

$$P_{\text{eff}}(r,t) = P_0(r) + \left\lfloor \frac{t - t_{\text{sub}}(r)}{\Delta_{\text{age}}} \right\rfloor, \quad \Delta_{\text{age}} = 50\,\text{ms} \tag{3}$$

with ties broken FIFO. Every accepted request is guaranteed *exact-once terminal settlement*: whether a job returns normally, raises a Common Lisp condition, or throws `'cancelled` at a `(checkpoint)`, $P_{\text{scheduler}}$ transitions its state to `:answered` once and emits `(done id status value)` via `credit-sink-put-control`.

4.4 Async-Signal-Safe Subprocess Multiplexing

External program execution (`src/exec.lisp`) illustrates why eliminating locks matters at the OS boundary. Spawning a child process requires simultaneously streaming `stdin`, draining `stdout` (preserving split multi-byte UTF-8 sequences across 64 KiB reads) and `stderr` (into a bounded 64 KiB circular `byte-ring`), enforcing deadlines, reacting to child termination (`SIGCHLD`), and aborting immediately on `cancel-token` firing without helper threads or sleep-polling.

Signaling an SBCL gate or mutex inside SBCL's `:status-hook` (which runs asynchronously upon `SIGCHLD`) is unsafe and risks lock re-entrancy deadlocks. We solve this with a lock-free, reference-counted **`wakeup-pipe`** whose entire lifecycle is packed into one 64-bit machine word (`state`):

$$\texttt{state} = (\text{writers}_{\text{in-flight}} \ll 32) \mid (\text{wfd} + 1) \tag{4}$$

where lower bits = 0 denote a closed write descriptor. Both the `SIGCHLD` status hook and the request's `cancel-token` hook invoke `wakeup-poke`: a CAS loop increments the upper 32-bit writer count if the lower 32 bits are non-zero, writes 1 byte via raw `unix-write` to the non-blocking pipe, and decrements the writer count—closing `wfd` if and only if a concurrent `close-wakeup-pipe` cleared the lower bits while the write was in flight. Consequently, a single `:io` worker multiplexes `[wakeup-rfd, stdin-fd, stdout-fd, stderr-fd]` inside one native `poll(2)` loop, waking in microseconds upon cancellation to send `SIGTERM` (followed after 1 s by `SIGKILL`) to the child's process group.

4.5 Frame-Clock Time-Sliced Host Ingestion

Isolating work in `lispm-kernel` is necessary but insufficient: if Emacs parsed thousands of incoming `chunk` lines inside its C process filter, redisplay would still stutter. In `lispm-kernel.el`, the process filter performs *zero S-expression parsing*: it merely splits incoming strings on newline boundaries in $O(n)$ and pushes raw lines onto an `incoming` list. Actual parsing and callback execution happen in `lispm-kernel-dispatch`, which processes ready lines up to a strict slice budget (τ_{slice} = 3 ms, default `lispm-kernel-slice-seconds`) and yields immediately if Emacs detects pending keyboard or mouse input (`input-pending-p`). Under LispM Desktop, `lispm-kernel-scheduler` binds dispatch to the desktop frame clock at priority depth 10 (after terminal frame updates), achieving cooperative real-time scheduling without polling timers.

5 Split-Responsibility Subsystems

Because `lispm-kernel` runs in a separate address space and intentionally cannot read or mutate Emacs buffers, subsystems that span disk state and live editor state require careful co-design. We highlight three production subsystems.

5.1 Incremental Session Indexing and P_{cache}

LispM's sidebar, terminal session picker, and cross-agent message bridge continuously query session histories of Codex (`rollout-*.jsonl`) and Claude Code (`PROJECT/*.jsonl`, including nested subagent trees). Reading entire multi-megabyte transcripts on every refresh is prohibitive.

In `src/sessions.lisp`, `scan-sessions` bounds per-file I/O by reading only each transcript's 64 KiB head and 16 KiB tail (expanding to the first 1 MiB only when Codex tool definitions push the first user prompt past 64 KiB, and inspecting the final 256 KiB when the 16 KiB tail lacks a turn-boundary marker to compute `:turn-ended`). Every file is fingerprinted by a nanosecond-resolution POSIX `stat` stamp:

$$\sigma(f) = \langle \text{kind, size, ino, dev, } t^{\text{ns}}_{\text{mtime}}, t^{\text{ns}}_{\text{ctime}} \rangle \tag{5}$$

To share cached readings across concurrent scans without read/write locks, a dedicated CSP custodian process, P_{cache}, maintains immutable `cache` snapshots tagged with monotonic epochs. At scan start, a worker requests the current immutable snapshot via `agent-cache` in $O(1)$ and fans out across four `work-group` threads with zero lock contention; unchanged files matching $\sigma(f)$ hit in memory without opening the file. At scan completion, the worker sends delta updates and the seen-file set to P_{cache}, which reconciles them against the base epoch and publishes a new snapshot while garbage-collecting deleted transcripts. On Emacs's side, views render immediately from the latest cached snapshot (zero blocking wait) and update reactively when background refreshes arrive; single-hop SSH projects automatically stage and talk to a remote `lispm-kernel` instance over SSH pipes.

5.2 Dirty-Buffer-Coherent Parallel Search

Agent codebase search (`search_files`) exemplifies split-state coherence. Offloading regex search blindly to an external tool (such as `ripgrep`) searches disk files rather than modified Emacs buffers, violating LispM's live-buffer invariant. Conversely, searching all files inside Emacs stalls the UI.

LispM partitions the search across both runtimes (`src/search.lisp`):

1. Emacs enumerates project files, collects the paths of buffers currently open with unsaved modifications (F_{open}), and submits (`:files` F `:pattern` p `:open-files` F_{open}) to `lispm-kernel`'s `:cpu` pool.

2. Inside `lispm-kernel`, a custom ripgrep-compatible regex engine scans all closed regular files ($F \setminus F_{\text{open}}$) in parallel across `work-group` threads, while immediately returning (`:file` f `:open t`) placeholders for $f \in F_{\text{open}}$ without touching disk.

3. Emacs merges the kernel's parallel disk matches with in-memory searches over just the small set F_{open} (and any remote TRAMP buffers), delivering multi-core speed with 100% live-buffer fidelity.

5.3 Non-Blocking Downloads, HTTP Streaming, and Verified Prebuilts

Finally, `lispm-kernel` eliminates network and fork stalls across three remaining paths: (1) `lispm-kernel-package-mode` ad-

vises `package.el`'s `package--with-response-buffer-1`, offloading background archive downloads to `fetch` workers running `curl` off-thread while preserving Emacs's GPG signature verification; (2) `http-stream` streams SSE model API responses via `curl` using `wakeup-pipe` multiplexing without Emacs process-filter churn; and (3) `lispm-kernel-program.el` and `lispm-prebuilt` distribute platform binaries whose SHA-256 digests are pinned inside git-tracked Elisp source (`*-sums.eld`) rather than fetched from mutable release metadata, falling back transparently to pure Elisp whenever no verified binary is available.

5.4 Implementation Footprint

LispM is implemented as modular open-source packages under AGPL-3.0-or-later. The host runtime (`lispm-agent`) comprises 40,792 lines of lexical-binding Emacs Lisp across 58 modules, including `lispm-runtime.el` (3,738 LoC), `lispm-tools.el` (3,247 LoC), `lispm-chat.el` (2,202 LoC), `lispm-core.el` (1,236 LoC), `lispm-transcript.el` (1,130 LoC), `lispm-lisp.el` (977 LoC), `lispm-ground.el` (937 LoC), `lispm-agents-bridge.el` (739 LoC), and `lispm-collab.el` (706 LoC). The sidecar (`lispm-kernel` v0.9.0) totals 4,604 lines of SBCL Common Lisp—`csp.lisp` (407 LoC), `wire.lisp` (276 LoC), `runtime.lisp` (763 LoC), `exec.lisp` (521 LoC), `search.lisp` (642 LoC), `sessions.lisp` (949 LoC), and JSON/-filesystem support (1,046 LoC)—paired with 2,857 lines of host Emacs Lisp adapters and 3,834 lines of unit and cross-runtime integration tests. All Elisp modules byte-compile cleanly with `byte-compile-error-on-warn` enabled, and SBCL builds fail on any compiler warning.

6 Evaluation

We evaluate both layers of LispM across four research questions:

- **RQ1 (End-to-End UI Responsiveness & Real Traces):** How effectively does the lock-free CSP sidecar eliminate main-thread stalls and GC pauses compared to pure Elisp and our v0.8 mutex sidecar across 12-run synthetic suites (0 and 8 active terminals) and a real multi-hundred-megabyte developer session trace?
- **RQ2 (CSP Kernel Scalability & Tail Latency):** How does $P_{\text{scheduler}}$ scale with concurrency, how fast do targeted cancellations wake blocked gates and `poll(2)` subprocesses, and does `credit-sink` bound memory without delaying control frames?
- **RQ3 (Subprocess Bursts, Env Ablation & Coherent Search):** How much stall does offloading subprocess bursts (`exec`) and hybrid repository searches (`search-files`) eliminate, and how much of the remaining submission cost stems from environment marshaling?
- **RQ4 (In-Image Harness Efficiency & Context Scaling):** How many LLM round-trips and context tokens does `run_lisp` save over sequential JSON tools, what is the runtime cost of deterministic replay continuation across M human approvals, and how does incremental prefix-folded Org parsing scale up to 120k-token contexts?

Experimental Setup. All benchmarks execute on an Apple M1 Max (10 cores, arm64, macOS) with GNU Emacs 31.1 (`-Q --batch`, default 800 KB `gc-cons-threshold`) and SBCL 2.6.9 (`lispm-kernel` v0.9.0 compiled via `save-lisp-and-die`, 40.6 MB standalone image). A 2.0 ms Emacs probe timer measures main-thread scheduling lateness $\Delta t = \max(0, t_{\text{actual}} - t_{\text{scheduled}})$. Session-discovery suites run **12 independent processes** per configuration (6 `elisp-first` and 6 `kernel-first` alternating pairs to neutralize OS page-cache warmth), reporting medians, interquartile ranges (IQR, [p25, p75]), and absolute worst-case bounds. All benchmark scripts and anonymized JSON artifacts are included in the repository.

6.1 RQ1: Session Discovery and Real-World Traces

Figure 2 and Table 2 evaluate session discovery on both the 256-file synthetic corpus (68.6 MB JSONL, where `lispm-kernel` reads 37.1 MB vs. Elisp's 20.3 MB to additionally compute `:turn-ended` for unended long sessions) and a real, privacy-anonymized developer workstation trace containing 203 Codex session files totaling **314.73 MiB** of JSONL (with individual transcripts reaching **107.99 MiB**).

Across 12 alternating runs, pure Elisp stalls Emacs continuously for the entire cold scan: median max stall is **194.97 ms** (IQR [194.1, 196.8] ms) when idle, **196.31 ms** (IQR [195.5, 202.2] ms) alongside eight active Rust Alacritty terminals streaming ∼2.4 MiB/s, and **311.92 ms** on the real 314.7 MiB trace (dropping 19 frames at 60 Hz / 37 frames at 120 Hz). Replacing v0.8's internal mutexes with v0.9's lock-free CSP channels cuts median max UI stall from 5.42 ms to **0.48 ms** (IQR [0.32, 0.57] ms, **405× lower** than Elisp and **11.3× lower** than v0.8) when idle, and from 6.64 ms to **0.54 ms** (IQR [0.40, 0.68] ms, **366× lower** than Elisp and **12.4× lower** than v0.8; 2.65 ms worst single sample across all 12 runs) under 8-terminal load, while improving 8-terminal cold wall time by 1.18× (239.51 ms vs. 283.74 ms) with **zero host Emacs GCs**. On the real 314.7 MiB trace, `lispm-kernel` additionally resolves nested Codex subagent trees (164 total sessions) while capping max UI stall at **0.25 ms** (**1,268× lower** than Elisp), and subsequent warm scans hit all 203 nanosecond `stat` stamps in P_{cache} in 11.33 ms (0.28 ms max stall).

6.2 RQ2: CSP Scheduler Scaling, Cancellation, and Backpressure

Figure 3 measures full wire-to-wire round-trip execution (`handle-line` parse → `control-mbox` → $P_{\text{scheduler}}$ → worker → `credit-sink` → P_{writer} format) across concurrency $C \in \{1, 4, 16, 64\}$ and 8 owners. Median round-trip latency at $C = 1$ is **16 μs** (47.1 kreq/s), scaling via single-owner batching to **142.4 kreq/s** ($C = 16$) and **150.0 kreq/s** ($C = 64$, p99 **628 μs**).

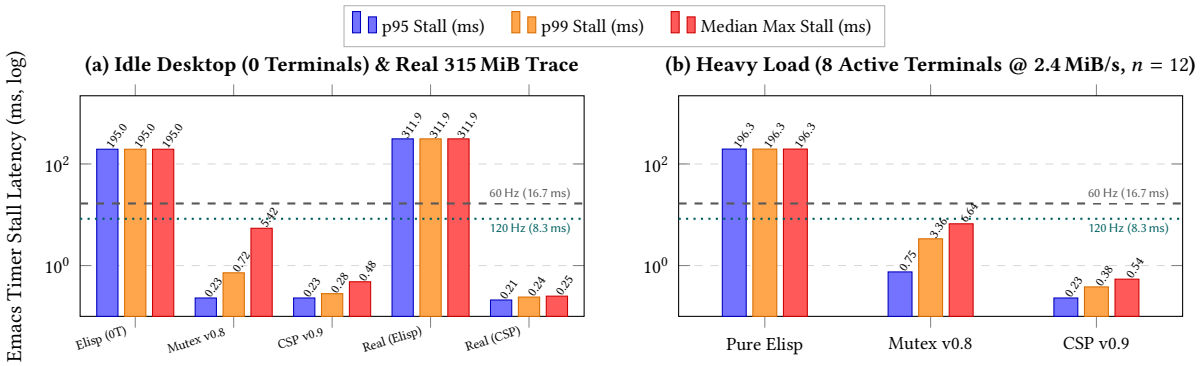


Figure 2: Cold session-discovery stall quantiles across 12 alternating runs of 256 synthetic transcripts (68.6 MB) and an anonymized 203-file real developer trace (314.7 MiB, max file 108.0 MiB). CSP v0.9 drives median max stall to **0.48 ms** idle (**405× lower**), **0.54 ms** under 8-terminal load (**366× lower** than Elisp; **12.4× lower** than Mutex v0.8), and **0.25 ms** on the real trace (**1,268× lower**).

Table 2: Session-discovery performance and Emacs 2 ms timer stall quantiles across **12 alternating runs** of 256 synthetic Codex transcripts (68.6 MB JSONL) plus an anonymized **203-file real developer workstation trace** (314.73 MiB logical JSONL, max single transcript 107.99 MiB). Bracketed ranges denote interquartile ranges [p25, p75] across the 12 runs.

Workload / Load	Runtime Mode	Wall Median [IQR] (ms)	p50 Stall	p95 Stall	p99 Stall	Max Stall Median [IQR] (ms)	Emacs GCs	Emacs GC (ms)
Synthetic 256 0 Terminals ($n = 12$ runs)	Pure Elisp (Cold)	196.95 [196.0, 198.7]	0.07 ms	194.97 ms	194.97 ms	194.97 [194.1, 196.8]	4.0	56.66
	Mutex Sidecar v0.8 (Cold)	210.48 [209.3, 212.1]	0.00 ms	0.23 ms	0.72 ms	5.42 [5.12, 8.93]	0.0	0.00
	CSP Sidecar v0.9 (Cold)	**226.90 [224.0, 231.4]**	**0.00 ms**	**0.23 ms**	**0.28 ms**	**0.48 [0.32, 0.57]**	**0.0**	**0.00**
	Pure Elisp (Warm)	2.64 [2.61, 2.74]	0.02 ms	0.65 ms	0.65 ms	0.65 [0.62, 0.75]	0.0	0.00
	CSP Sidecar v0.9 (Warm)	**4.17 [3.86, 4.29]**	**0.09 ms**	**0.31 ms**	**0.31 ms**	**0.31 [0.25, 0.43]**	**0.0**	**0.00**
Synthetic 256 8 Terminals (~2.4 MiB/s, $n = 12$)	Pure Elisp (Cold)	198.29 [197.4, 204.2]	0.03 ms	196.31 ms	196.31 ms	196.31 [195.5, 202.2]	4.0	60.01
	Mutex Sidecar v0.8 (Cold)	283.74 [278.5, 294.0]	0.00 ms	0.75 ms	3.36 ms	6.64 [6.59, 6.70]	0.0	0.00
	CSP Sidecar v0.9 (Cold)	**239.51 [236.2, 260.4]**	**0.00 ms**	**0.23 ms**	**0.38 ms**	**0.54 [0.40, 0.68]**	**0.0**	**0.00**
	Pure Elisp (Warm)	2.34 [2.23, 2.45]	0.04 ms	0.43 ms	0.43 ms	0.43 [0.41, 0.48]	0.0	0.00
	CSP Sidecar v0.9 (Warm)	**6.87 [6.45, 20.7]**	**0.03 ms**	**0.32 ms**	**0.32 ms**	**0.32 [0.27, 0.43]**	**0.0**	**0.00**
Real Dev Trace 203 Files, 314.7 MiB (max file 108.0 MiB)	Pure Elisp (Cold)	313.89	0.06 ms	311.92 ms	311.92 ms	311.92	3.0	40.12
	CSP Sidecar v0.9 (Cold)	**436.79**	**0.00 ms**	**0.21 ms**	**0.24 ms**	**0.25**	**0.0**	**0.00**
	CSP Sidecar v0.9 (Warm)	**11.33**	**0.03 ms**	**0.28 ms**	**0.28 ms**	**0.28**	**0.0**	**0.00**

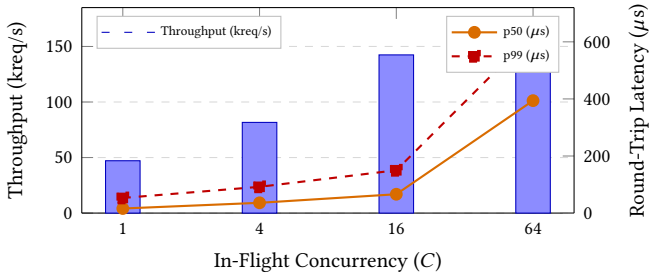


Figure 3: Full wire-lexer + lock-free CSP $P_{\texttt{scheduler}}$ + worker + `credit-sink` round-trip scaling across concurrency $C \in \{1, 4, 16, 64\}$. Throughput reaches **150.0 kreq/s** at $C = 64$ while keeping p99 end-to-end latency at **628 µs**.

Table 3 confirms our reactivity and backpressure bounds: (1) cancelling a blocked `wait` gate settles on the wire in **27 µs** (p50) and **85 µs** (p99); (2) cancelling an external `/bin/sh` child blocked in `poll(2)` wakes via the 64-bit CAS `wakeup-pipe`, sends process-group `SIGTERM`, reaps `SIGCHLD`, and settles on the wire in **0.58 ms** (p50) and **1.12 ms** (p99); and (3) streaming 16 MiB into a stalled 4 MiB `credit-sink` halts producers at **exactly 4,194,304 bytes** (zero overshoot) while 200 concurrent control frames bypass the saturated sink in **≤1.0 µs** (p99).

Table 3: Microbenchmark of `lispm-kernel` v0.9 CSP cancellation wakeup and `credit-sink` backpressure invariants.

CSP Mechanism / Metric	p50	p95	p99	Max / Bound
In-memory `cancel-token` gate ($n = 200$)	27 µs	69 µs	85 µs	100 µs
`poll(2)` `wakeup-pipe` + `SIGTERM` ($n = 50$)	0.58 ms	0.68 ms	1.12 ms	1.12 ms
`credit-sink` control bypass ($n = 200$)	<1 µs	1.0 µs	1.0 µs	9.0 µs
`credit-sink` peak queued (16 MiB stream)	—	—	—	**4,194,304 B**

6.3 RQ3: Subprocess Bursts, Env Ablation, and Search

Figure 4 and Table 4 evaluate a 32-process `git rev-parse HEAD` burst and a repository regex search across 202 real LispM `.el` files (6.47 MB, 6,231 definitions, 3 open buffers).

Even Emacs's asynchronous `make-process` stalls the main thread for **69.45 ms** (>4 frames at 60 Hz) due to main-thread descriptor/fork setup across 32 processes. Offloading via `lispm-kernel-exec` drops steady-state p95 stall by **298×** to **0.23 ms** (wall time rises to 946 ms because

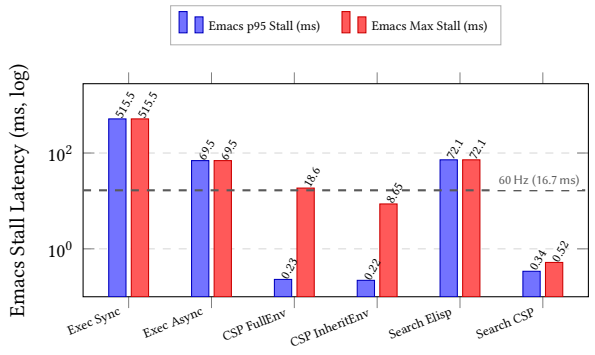


Figure 4: Main-thread p95 and max stall latency under a 32-process `git` burst (comparing `call-process`, Emacs async `make-process`, `lispm-kernel-exec` with per-call `process-environment` marshaling, and environment-inheriting submission) and 202-file (6.47 MB) coherent regex search.

Table 4: Subprocess burst (32 × `git rev-parse`, including environment-inheritance ablation) and coherent regex search across 202 real LispM `.el` files (6.47 MB, 6,231 matches, 3 dirty open buffers; medians of 4 runs).

Workload	Mechanism	Wall (ms)	p95 Stall	Max Stall	GCs (ms)
32× `git` Burst	Sync `call-process`	517.46	515.50 ms	515.50 ms	0 (0.0)
	Async `make-process`	87.43	69.45 ms	69.45 ms	0 (0.0)
	CSP exec (Full Env)	946.17	0.23 ms	18.55 ms	2 (16.7)
	CSP exec (Inherit Env)	**917.03**	**0.22 ms**	**8.65 ms**	**1 (8.7)**
202-File Search	Elisp Sync Search	74.09	72.12 ms	72.12 ms	0 (0.0)
	CSP `search-files`	**68.67**	**0.34 ms**	**0.52 ms**	**0 (0.0)**

`lispm-kernel` bounds `:io` concurrency to 4 child processes to prevent fork storms). Our ablation confirms the exact source of the single 18.55 ms max spike: serializing Emacs's full `process-environment` 32 times in an un-yielded loop triggers two GCs (16.72 ms) under batch Emacs's 800 KB `gc-cons-threshold`. Passing `:env nil :env-inherit t` (*Inherit Env*) cuts wire allocation in half, reducing max stall to **8.65 ms** (a single 8.71 ms GC; < 0.3 ms outside GC). On the 202-file search, hybrid `search-files` finishes faster end-to-end (68.67 ms vs. 74.09 ms) while cutting p95 stall by **211×** (**0.34 ms** vs. 72.12 ms, 0.52 ms max), returning 6,142 matches from 199 closed files and delegating the 3 open buffers (`:open t`) back to Emacs for 100% live-buffer accuracy.

6.4 RQ4: In-Image Harness Overhead, Replay Continuation, and Context Scaling

Finally, Figure 5 and Table 5 evaluate the host-side mechanisms of §3 directly inside GNU Emacs:

1. **Deferred Tool Schema Compression:** LispM's 25-tool core session schema occupies 20.4 KB (~5,101 tokens), compared to 35.0 KB (~8,741 tokens) when eagerly exposing all 44 built-in and Lisp-image tools (41.6% reduction before counting external MCP catalogs such as a 26-tool GUI server at ~14,000 tokens, yielding > 63% total schema savings).

Table 5: In-image harness evaluation: (top) 40-file selective refactoring ($M \in \{1, 5, 10\}$ human write approvals) comparing sequential JSON tool turns vs. `run_lisp` replay continuation; (bottom) Org context buffer round-trip and parse latency.

40-File Refactor (M Edits)	**LLM Turns**	**Prompt Payload**	**Disk Reads**	**Replay Passes**	**Local Time**
Seq. JSON Tools ($M = 1/5/10$)	41 / 45 / 50	~45.2–45.5k tok	40	1	32.1–49.6 ms
`run_lisp` ($M = 1$)	**1**	**254 tok**	**40**	2	**25.1 ms**
`run_lisp` ($M = 5$)	**1**	**254 tok**	**40**	6	**44.1 ms**
`run_lisp` ($M = 10$)	**1**	**255 tok**	**40**	11	**62.1 ms**
Org Context Size	**Messages**	**Byte-Exact**	**Cold Parse**	**Incr. Append**	**Speedup**
11.0k tok (43.9 KB)	23	100%	3.61 ms	**0.66 ms**	**5.5×**
50.8k tok (203.0 KB)	103	100%	16.90 ms	**1.65 ms**	**10.2×**
120.4k tok (481.5 KB)	243	100%	39.63 ms	**3.49 ms**	**11.3×**

2. **`run_lisp` Token Savings & Replay Efficiency:** On a 40-file (160 KB) Emacs Lisp refactoring benchmark where $M \in \{1, 5, 10\}$ files match a target header under policy `((read . auto) (write . ask))`, sequential JSON tool calling requires 41–50 LLM round-trips and accumulates 180.6–181.8 KB (~45.2–45.5k tokens) of file contents in the prompt. `run_lisp` completes the entire workflow in **1 LLM round-trip** and **1,020 bytes (~255 tokens,** 178× **smaller)** by filtering inside the pure Lisp loop. Crucially, across $M = 10$ sequential human approval suspensions (11 total execution passes), memoized replay executes the underlying `read_file` effect **40 times total** (never $40 \times 11 = 440$) and `edit_file` exactly 10 times, adding only ~3.7 ms per replay pass (62.1 ms total across all 11 passes).

3. **Prefix-Folded Org Context Scaling:** Across 11k, 51k, and 120k-token Org transcripts containing comma-escaped Org directives and preserved `thinking` signatures, `lispm-transcript-parse` achieves 100% byte-exact round-trip fidelity. While cold full-buffer parsing scales linearly to 39.63 ms at the 120k-token compaction ceiling (243 messages), incremental prefix folding (`lispm-transcript--folded`) parses normal turn appends in **0.66 ms** (11k), **1.65 ms** (51k), and **3.49 ms** (120k)—**11.3× faster** and well inside a single 60/120 Hz frame.

7 Discussion, Related Work, and Conclusion

7.1 Limitations and Future Directions

Our evaluation highlights four boundaries of the current architecture:

1. **Host Callback Serialization:** Because `lispm-kernel` never mutates Emacs memory directly, final UI updates and buffer edits execute on Emacs's thread in 3 ms slices. A single poorly written callback exceeding 3 ms cannot be preempted mid-form.

2. **Cross-Platform & End-to-End Display Timing:** Batch-mode 2 ms timer lateness accurately captures main-thread evaluation and GC stalls on macOS, but

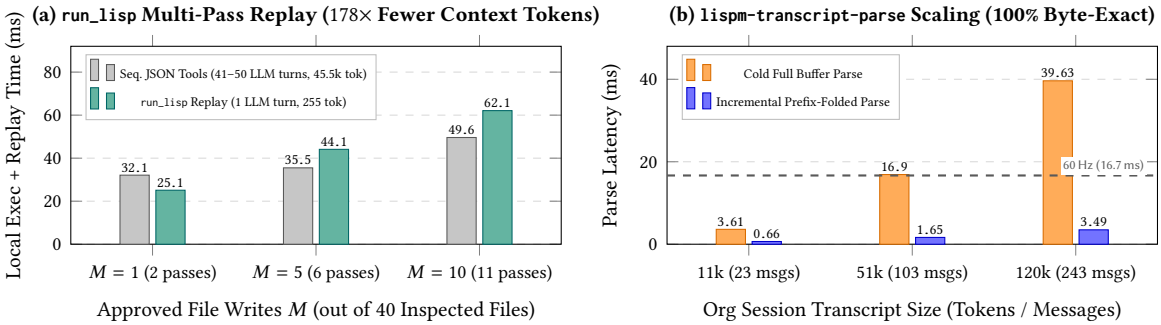


Figure 5: In-image harness evaluation (§6.4): **(a)** Refactoring 40 files with $M \in \{1, 5, 10\}$ human write approvals via `run_lisp` collapses 41–50 LLM round-trips and ~45.5k tool-output tokens into **1 round-trip** and **255 tokens** (178× smaller) while adding only ~3.7 ms per deterministic replay pass (40 memoized reads executed once). **(b)** Incremental prefix-folded Org parsing (`lispm-transcript--folded`) keeps turn-append parsing at **3.49 ms** (11.3× faster than cold re-parse) even at the 120k-token compaction threshold.

omits window-system GPU compositing and Linux kernel scheduling (`poll(2)`/pipe buffering under glibc). Extending our automated suite to Linux `x86_64`/`arm64` runners and measuring end-to-end input-to-photon latency under Emacs 32 OpenGL canvases (`lispm-gl`) are natural next steps.

3. **Environment Generation Interning:** Our ablation in §6 proved that inheriting the sidecar's environment cuts 32-process burst submission max stalls from 18.55 ms to 8.65 ms (a single GC under `emacs -Q`'s 800 KB threshold); interning modified `process-environment` deltas by generation ID in `lispm-kernel-exec` will extend this zero-marshaling fast path to custom buffer-local environments automatically.

4. **Static Purity Allowlists in `run_lisp`:** `run_lisp` relies on Emacs's `side-effect-free` declarations rather than a formal type-and-effect proof; generic functions (`seq-*`) dispatching on custom structs could still invoke arbitrary methods unless restricted.

7.2 Related Work

Lisp Machines and Malleable Environments. CONS/CADR [4], Symbolics Genera [15, 21], and Smalltalk-80 [9] pioneered unified live-object workspaces, typed screen presentations, and condition/restart error recovery [18]. GNU Emacs [14, 20] carried Lisp-image extensibility onto mainstream POSIX systems. LispM revives Genera's presentation, inspector, and condition-restart idioms specifically as human-agent interaction primitives while overcoming Emacs's single-threaded runtime ceiling via a modern multi-core Common Lisp sidecar.

Split Kernels, Event Stages, and Lock-Free Concurrency. Barrelfish's Multikernel [2] and Helios [16] demonstrated that explicit message passing across isolated cores or satellite kernels scales more predictably than shared-memory locks. SEDA [23] combined event-driven stages with bounded queues for overload conditioning. `lispm-kernel` synthesizes these principles with Hoare's CSP [7, 8], Michael–Scott lock-free queues [13], and work-helping pools [3], adding two domain-specific mechanisms for interactive hosts: a byte-budgeted `credit-sink` with wait-free control bypass, and a 64-bit CAS `wakeup-pipe` unifying `SIGCHLD` delivery with CSP cancellation. Whereas CPython PEP 703 [5] removes the GIL through invasive C runtime surgery, our split-Lisp design achieves multi-core parallelism and GC isolation with zero modifications to Emacs's C core.

LLM Agent Harnesses and Operating Systems. ReAct [25], SWE-bench/SWE-agent [10, 24], and CodeAct [22] established iterative tool and code execution loops for software engineering agents. MemGPT [17] and AIOS [12] proposed OS-inspired memory paging and LLM syscall scheduling inside standalone Python daemons. Unlike external harnesses or MCP-only clients [1], LispM embeds agents directly inside the user's live editing image, unifying unsaved buffers, Org-mode editable contexts, replayable effect-mediated Lisp code actions, and heterogeneous CLI agent collaboration.

7.3 Conclusion

AI agents do not need to scrape pixels or tunnel through dozens of brittle wrapper tools when the host environment is already a self-documenting Lisp image. LispM demonstrates that pairing GNU Emacs's stateful live-buffer runtime with a crash-isolated, lock-free CSP Common Lisp sidecar (`lispm-kernel`) delivers both halves of the Lisp Machine vision on modern multi-core hardware: unified in-image agency and sub-millisecond interactive responsiveness.

References

[1] Anthropic. Model context protocol (MCP) specification. https://modelcontextprotocol.io/specification, 2024.

[2] A. Baumann, P. Barham, P.-E. Dagand, T. Harris,

R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, pages 29–44. ACM, 2009. doi: 10.1145/1629575.1629579.

[3] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999. doi: 10.1145/324133.324234.

[4] R. D. Greenblatt, T. F. Knight, Jr., J. T. Holloway, and D. A. Moon. A LISP machine. *SIGIR Forum*, 15(2):137–138, 1980. Also MIT AI Memo 444 (CONS/CADR), 1977.

[5] S. Gross. PEP 703: Making the global interpreter lock optional in CPython. In *Python Enhancement Proposals*, 2023. Python Software Foundation.

[6] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 235–245, 1973.

[7] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978. doi: 10.1145/359576.359585.

[8] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[9] D. H. H. Ingalls. Design principles behind Smalltalk. *BYTE Magazine*, 6(8):286–298, 1981.

[10] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *Proceedings of the 12th International Conference on Learning Representations (ICLR)*, 2024.

[11] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, pages 611–626. ACM, 2023. doi: 10.1145/3600006.3613165.

[12] K. Mei, Z. Li, S. Xu, R. Ye, Y. Ge, and Y. Zhang. AIOS: LLM agent operating system. In *Proceedings of the First Conference on Language Modeling (COLM)*, 2024.

[13] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 267–275. ACM, 1996. doi: 10.1145/248052.248106.

[14] S. Monnier and M. Sperber. Evolution of Emacs Lisp. *Proceedings of the ACM on Programming Languages*, 4 (HOPL):1–55, 2020. doi: 10.1145/3386324.

[15] D. A. Moon. Architecture of the Symbolics 3600. *ACM SIGARCH Computer Architecture News*, 13(3):76–83, 1985. doi: 10.1145/327070.327133.

[16] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, pages 221–234. ACM, 2009. doi: 10.1145/1629575.1629597.

[17] C. Packer, S. Wooders, K. Lin, V. Fang, S. G. Patil, I. Stoica, and J. E. Gonzalez. MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

[18] K. M. Pitman. Condition handling in the Lisp language family. In *Advances in Exception Handling Techniques*, volume 2022 of *Lecture Notes in Computer Science*, pages 39–59. Springer, 2001. doi: 10.1007/3-540-45407-1_3.

[19] C. Rhodes. SBCL: A sanely-bootstrappable Common Lisp. In *Self-Sustaining Systems*, volume 5146 of *Lecture Notes in Computer Science*, pages 74–86. Springer, 2008. doi: 10.1007/978-3-540-89275-5_5.

[20] R. M. Stallman. EMACS: The extensible, customizable self-documenting display editor. In *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, pages 147–156. ACM, 1981. doi: 10.1145/800209.806466.

[21] J. H. Walker, D. A. Moon, D. L. Weinreb, and M. McMahon. The Symbolics Genera programming environment. *IEEE Software*, 4(6):36–45, 1987. doi: 10.1109/MS.1987.232087.

[22] X. Wang, Y. Chen, L. Yuan, Y. Zhang, Y. Li, H. Peng, and H. Ji. Executable code actions elicit better LLM agents. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, pages 50208–50232, 2024.

[23] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 230–243. ACM, 2001. doi: 10.1145/502034.502057.

[24] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. R. Narasimhan, and O. Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, 2024.

[25] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao. ReAct: Synergizing reasoning and acting in language models. In *Proceedings of the 11th International Conference on Learning Representations (ICLR)*, 2023.