Attention Follows the Work: Foveated Transformer Worktrees for Coding Agents

Peter Zatloukal *
agent-hypervisor.ai/papers
peter@zatloukal.com

Abstract

Transformer-based coding agents read a small, predictable part of a repository and spend as if they did not. Across 498 agent sessions on eighteen repositories and three machines, the median session touches 2 to 9% of a repository's files, depending on the machine, and in every repository with five or more sessions the top decile of the files read takes at least 41% of the reads, four times its share under uniform reading. The region is predictable before the task starts: the ten most recently committed files name a mean 46 to 68% of the pre-existing files a task will edit and 32 to 50% of those it will read. Yet the agent spends the same model, the same reasoning effort, and the same full-text context on every file it opens. We argue this is the situation foveated rendering solved for virtual reality: predict where the eye will land, render that region fully, and render the periphery cheaply. We introduce FTW, which treats the agent's file-level *gaze* as a predictable, learnable signal: a saliency predictor ranks files from cheap repository signals and the agent's first accesses; a *foveated worktree* presents the predicted region at full text and the rest as signatures or names, with on-demand materialization logged as *saccades*; a compute controller gives the frontier model and high effort to edits and verification inside the fovea and cheaper tiers to the periphery; and a risk term keeps scrutiny on central and security-sensitive code regardless. Two things are measured beyond these sessions. On all 500 SWE-bench Verified instances, git recency recovers 0.11 of the gold edit set at ten files against 0.46 to 0.68 on one developer's machines, so the cheapest pre-run signal is a property of the regime, not of code; on the same instances a small model names 0.90 of the gold files from the issue text alone, no checkout, so the benchmark's edit set is nameable without the repository. And a three-task pilot of the mechanism, with hidden tests and three seeds, passed all 36 runs at 16% lower cost with an oracle fovea and 6% with a predicted one, a saving a byte accounting attributes to shorter trajectories, since file reads are 14% of the context processed in the pilot machine's sessions; fifteen single-file demotions and ten adversarial tasks flipped no hidden test, and the agent materialized every misranked file before its first edit. Everything else in the results is a pre-registered target, marked with a dagger and written before the experiment that decides it: on SWE-bench Verified, FTW matching uniform allocation within $0.4^{\dagger}$ points of pass@1 at $58\%^{\dagger}$ of the input tokens, $96\%^{\dagger}$ of the all-frontier pass rate at $45\%^{\dagger}$ of its cost, a wrong fovea costing $3.1^{\dagger}$ points and $21\%^{\dagger}$ more spend rather than failure, and a predictor that improves with a repository's own traces. The objective throughout is task success under a compute budget; gaze prediction is the instrument, and the paper says how the loop is kept from learning the agent's blind spots.

*ORCID 0009-0000-0451-0591; Google Scholar profile.

Preprint. Working-backwards draft (2026-11-02, revision 3). Numbers marked † are pre-registered targets, not measurements; the rest are measured or quoted from the literature.

1 Introduction

Foveated rendering exploits an asymmetry: the eye resolves fine detail only in the central few degrees of the visual field, so a renderer that knows where the eye is pointed can shade that region fully and the rest coarsely. Guenter et al. reported shading 10 to 15 times fewer pixels with no perceptible loss [21]; gaze-tracked headsets and gaze predictors made the trick practical [23, 51]. The method needs two things: a predictor of where attention will go, and a pipeline that can vary resolution by region.

Coding agents, the transformers that now write and maintain a growing share of software, have the same asymmetry and neither of the two things. An agent given a task in a repository of a thousand files will read a few dozen and edit a handful, and it will find them by *progressive orientation*: a harness injects a map (a `CLAUDE.md` or `AGENTS.md`, the git status), the agent lists directories, opens a few candidate files, searches for identifiers, and narrows until it edits. Everything it opens arrives at full text. Every reasoning step runs on the same model at the same effort setting, whether it is fixing the function the task is about or confirming that a utility file is irrelevant. In the sessions we measured, the frontier model handled 100% of the spend; cheaper tiers were used only for a one-line documentation lookup.

A word on the title: attention in the transformer [71] is the mechanism by which the model weights every token it is given against every other. Attention in this paper is the agent's, in the behavioral sense: where it looks, measured at the harness's tool boundary by the files its calls open (Section 3), which is the mechanism by which a harness decides what the model is given at all. The two are different questions. The model's internal attention tracks human programmers' attention poorly [34]; what the agent reads is observable in its tool calls, is what costs money, and is what a harness can allocate. We study the second, downstream of the transformer and of the coding-agent primitive built on it, and the transformer in the title is the agent whose reading we foveate.

This paper asks whether the agent's gaze over a worktree can be predicted and exploited the way a renderer exploits the eye. The pilot's short answer, and the paper's title, is that attention follows the work: the files an agent reads concentrate on the files it will edit, nearly every edit is preceded by a read, and both sets are predictable from the repository's own history and from the agent's first few accesses. Three questions follow, and the paper is organized around them.

1. **Is gaze concentrated?** Section 3 reports a descriptive study of 498 Claude Code sessions on eighteen repositories across three of the developer's machines, with file-level access extracted from tool calls, including reads hidden inside shell commands. Within a session, the median share of the repository touched is 2.2% to 8.6% depending on the machine; within each repository with five or more sessions, the top decile of the files read takes 41 to 77% of the reads.
2. **Is it predictable?** Before a task runs, cheap signals already say a lot: on these single-developer repositories, the ten most recently committed files recover 46 to 68% of the files the task will edit and 32 to 50% of the files it will read, depending on the machine. On the pilot, after the agent's first five distinct reads, the agent's own trajectory recovers 45%. Predicting the *edit* set is a well-studied problem that file-level localizers now handle well; the target that matters for cost is the *access* set, the files the agent reads to decide, which in the pilot is six times larger (median 18 files read against 3 edited) and which no localizer predicts. Section 4 builds a predictor for both, and Section 6 measures it on SWE-bench Verified, where recency is far weaker.
3. **Can it be exploited?** FTW[2] varies three resources by predicted saliency: context resolution (full text, signatures, names), model tier, and reasoning effort, with verification effort following a risk term. Section 6 reports cost-matched and success-matched comparisons against uniform allocation, and the cost of being wrong.

Contributions. (i) A formulation of a coding agent's attention as a *gaze trace* over a worktree, with two metrics that let predictors be compared and allocations be budgeted: the foveation curve and the saccade rate (Section 3). (ii) A descriptive study of agent gaze on real development sessions, with an extractor that recovers file access from shell commands (Section 3, Appendix A). (iii) FTW: a saliency predictor conditioned on orientation, a foveated worktree with logged saccades, a compute controller over model and effort tiers, and a risk term (Section 4). (iv) Pre-registered experiments on SWE-bench Verified and a fresh suite with hidden tests, including the *acuity curve* of a language

[2] We also admit the obvious third expansion.

model (accuracy against resolution tier by graph distance from the edit), the cost of a wrong fovea, and a learning curve over repository traces (Sections 5–6). (v) A release of the extractor, the worktree mechanism, and the gaze dataset.

None of the parts is new on its own. Repository maps, file-level localization, degree-of-interest models, model routing, and selective verification each have precedents (Section 2). The contribution is the controller that joins them, specific to a repository and fit to the agent's own traces, together with the measurements that say whether it pays and the pre-registered criteria under which it fails.

2 Background and related work

Foveation in graphics and vision. Foveated rendering renders the gazed region at full resolution and the periphery coarsely, originally with three nested layers [21], later with contrast-preserving blur that tolerates coarsening within 30 degrees of the fovea [51], hardware variable-rate shading [46, 70], and neural reconstruction of the periphery from about a tenth of the pixels [27]. Without an eye tracker, gaze is predicted from scene saliency and head trajectory [12, 22, 23]. Two facts from vision science shape our design. Peripheral vision is not blurred central vision but a pooled summary of statistics over regions that grow with eccentricity [62], which is what a signature or a docstring is to a source file. And inhibition of return [32, 54] discourages revisiting a fixated location; agents, we find, revisit constantly (Section 3). On the model side, hard-attention and glimpse models allocate computation to a sequence of fixations [40, 58], foveated and peripheral vision transformers vary token resolution by eccentricity [25, 39], token pruning and merging spend per-instance budgets [4, 56, 80], and skim-then-read models do the same for text [45, 63]. One lesson transfers with a sign flip: DeepFovea may hallucinate the periphery because the criterion is perceptual, while a summary that invents an API breaks the build. FTW's periphery is therefore derived and checkable, never generated.

Attention inside the model. The transformer's attention [71] is itself a learned allocation of computation over its input, dense over the context and quadratic in its length, which is why context is the resource every agent harness rations and why a file that enters the context is paid for on every later call that attends over it. Foveation has been brought inside the architecture for vision: FoveaTer samples an image at a resolution that falls away from a predicted fixation and attends over the result [25]. Ours sits one level up, over what enters a coding agent's context, and leaves the attention inside the model as it is; the attention we measure is the agent's, behavioral (Section 3), and the savings we claim are in what the model is given, not in how it weighs it.

Adaptive compute and model routing. Test-time compute helps most when it is allocated by predicted difficulty rather than uniformly [10, 66]; models left to ration a shared budget across sub-problems do it badly [14], and in coding agents uniform deep reasoning is often wasted: selecting the less over-thinking of two candidates improved SWE-bench results by about 30% at 43% lower cost [9], and a cost-first leaderboard found higher reasoning effort reducing accuracy in most runs [28]. Cascades and routers choose a model per query [7, 11, 47]; for coding, SWE-Router lets a cheap model explore and a value head decide whether to escalate the whole task [67], and AgentRouter and Ares select the model tier and effort *per step* inside one trajectory, reporting cost cuts near 72% at under 3 points of loss [52, 76]. Fixed phase splits are the practitioner version: CodeMonkeys scans every file with a cheap model and edits with an expensive one [13], aider's architect mode and Claude Code's plan-with-Opus setting split planning from execution [17], and Minions split a document between a local and a remote model [42]. Architecture-level methods vary depth per token [20, 57]. These allocate along *time* (which step) or per query; FTW allocates along *space* (which region of the worktree), and the two compose. One constraint carries over: switching models mid-trajectory recovers under half of the quality gap because the transcript does not transfer [15], so FTW switches tiers at subagent boundaries and hands over repository state, not conversation. The product surface exists: per-request effort levels and per-subagent model overrides [1].

Localization and context construction for coding agents. Issue localization predicts the gold patch's files from the issue text, and at file level it is close to solved: retrieval localizers [79, 87], the hierarchical file-to-element-to-line ladder of Agentless [75], graph-guided agents [8, 49], and code rankers reach file accuracy@5 above 90% on SWE-bench Lite at cents per instance [59]. Two caveats bound those numbers: models name the buggy file from the issue text alone at 76% on SWE-bench Verified against 53% on less-exposed repositories, which is memorization [36], and

post-cutoff benchmarks exist for exactly this reason [2, 78]. The edit set, however, is not where the tokens go. Reads are 76% of an agent's tokens on SWE-bench Verified [74], trajectories inspect many times more files than they change [30], and whether the agent *opened* the right file separates resolved from unresolved runs (83% against 40% in one trajectory corpus) [5, 43]. Context construction addresses this reactively: repository-level retrieval [3, 85], aider's map of tree-sitter signatures ranked by PageRank [16], just-in-time and progressively disclosed context [55], and SWE-Pruner, which prunes what the agent has already read [74]. FTW predicts the *access set* ahead of the reads, conditions on the agent's own orientation as it unfolds, and learns from agent traces rather than gold patches, which no localizer we found does. Trajectory corpora supply the traces [24, 43, 73, 77], though none ships a per-step file-access table; it has to be parsed from logs, as in Section 3.2.

Human programmer attention. Programmers forage. Information foraging theory models navigation as following scent toward patches of value [53]; in debugging, scent-following predicts where programmers go better than their stated hypotheses do [35], and developers spend about 35% of a maintenance session on the mechanics of navigation, revisiting a small set of relevant fragments [33]. Eye tracking shows code reading departs from linear order (linearity near 80% on prose, 70% on code for novices, 60% for experts) [6], and human gaze has served as supervision for code models [61, 86], while a language model's internal attention is not aligned with where programmers look [34, 50], which is why we define attention behaviorally, by tool calls. The closest engineering precedent is Mylyn's degree-of-interest model, which accrued interest per interaction, decayed it per later event, and filtered the IDE to the task's context; programmers' edit-to-selection ratio rose for 13 of 16 in its field study [29]. ROSE mined version histories to predict co-changing files (26% precision and 15% recall at file level, with a correct location in the top three for 64% of transactions) [89], and topology analysis expanded from a few focus points through the dependency graph [60]; both are baselines in Section 5. Learned context selection for completion [65] and recently-open-tab heuristics [18] are the modern descendants. FTW is a degree-of-interest model for an agent, learned from the agent's traces, used to allocate compute rather than to filter a view; we found no prior work that builds one from agent traces or compares agent navigation with the human metrics above.

Where scrutiny belongs. Change and defects concentrate: the top fifth of files held 71 to 92% of faults across 26 industrial releases [48], a tenth of files held 73 to 95% of future faults in FixCache's locality model [31], churn predicts defect density [41], and dependency-network centrality adds signal over complexity [88]; hotspots (churn times complexity) locate maintenance risk [69], with the caveat that the Pareto shape depends on how multi-file fixes are counted and the top fifth is hard to pick from size and churn alone [72]. Vulnerabilities concentrate too: process metrics find over 80% of vulnerable files at under 25% false positives [44, 64], and 8% of Firefox files held 72% of its vulnerabilities [68]. Just-in-time risk models direct review effort at change time [26], and predictive test selection halves test cost while catching nearly all faulty changes [37]. We found no work that couples these predictors with a language model's compute or verification allocation; FTW's risk term does.

3 The gaze of a coding agent

3.1 Definitions

Let R be the set of files in a worktree. A *task* is one user request and the agent's work until the next. The agent's *gaze trace* is the ordered sequence $g = (a_1, \ldots, a_n)$ of file-level accesses it makes: a *read* shows a file's content to the model, a *search* scans a directory, an *execution* runs a repository file, an *edit* writes one. The *access set* $A \subseteq R$ is the set of distinct files in g and the *edit set* $E \subseteq A$ the distinct files edited. The *orientation* of a task is the prefix of g before its first edit.

Definition 1 (Fovea and foveation curve)**.** *Given a ranking ρ of R and a budget $b \in [0, 1]$, the fovea F_b is the top $\lceil b|R| \rceil$ files of ρ. The foveation curve of ρ for target $T \in \{A, E\}$ is $r_\rho(b) = |F_b \cap T|/|T|$. Its area,* AUFC*, is 1 for a ranking that puts T first and 0.5 in expectation for a random ranking.*

Definition 2 (Saccade)**.** *An access to a file outside F_b is a saccade. The saccade rate of a task is the number of saccades per task at budget b; the saccade log is the sequence of (task, file, position in trace) triples.*

Table 1: Concentration of agent gaze, pooled over all sessions in a repository. Tracked files per the repository's git index; *coverage* is the median share of tracked files a single session read, edited, or executed. Gini: the Gini coefficient of per-file read counts, 0 when every file read is read equally often and approaching 1 as one file takes nearly all reads; it is twice the area between the repository's Lorenz curve (Figure 1) and the diagonal. Measured (pilot).

Repository	Tracked	Sessions	Coverage	Gini	Top 10%	Top 20%	Reads on edited files
R1 static site (JS/MD)	106	38	5.7%	0.78	77%	86%	73%
R2 game server (Python)	1,039	7	0.4%	0.58	45%	66%	39%
R3 agent harness (Bash/MD)	45	5	64.4%	0.61	57%	71%	39%
R4 data generator (Python)	2,227	5	0.6%	0.71	66%	78%	32%
R5 ML pipeline (Python)	212	5	11.8%	0.64	48%	70%	49%
Pooled		60	5.7%	0.72	67%	80%	

The foveation curve is the Lorenz curve of a predictor's usefulness: a budget on the x axis, the share of the work it covers on the y axis. The saccade rate is the price of a predictor's errors in a system that materializes the periphery on demand, which is the system of Section 4.

How gaze is observed. Every access in g is a tool call. A coding agent reaches the worktree only through its harness: a read is a `Read` call or a shell command whose output is returned to the model, a search is a `Grep` or `Glob` call, an edit is an `Edit` or `Write` call, and the harness records each call with its arguments and its result. We measure gaze at that boundary, not at the disk, for three reasons. The tool result is what enters the model's context, so its size is the cost and its content is what the model can act on; a shell command that greps a file reads all of it and shows one line. The processes the agent launches, builds and tests, read most of the repository without showing the model any of it (Section 4.2), so a disk-level trace would conflate the agent's reading with its subprocesses'. And the tool boundary is where a harness can intervene: the hook of Section 4.2 sees every call before it runs, which is how a predicted fovea is enforced and how the agent's first k accesses reach the predictor as they happen. The cost of the choice is that reads hidden inside shell commands have to be recovered by parsing the command, which Appendix A does heuristically and which undercounts (reads inside interpreter one-liners are missed), and that in the intervention pilots the shell readers are disabled so that every read passes through the hook. Subagents' calls are recorded in their own transcripts and count toward the task that spawned them.

3.2 A descriptive study

The study spans three of the developer's machines, 498 sessions in all, 470 of them with file access (Table 4). This section analyzes the first, the pilot, in depth and then sets the other two beside it. On the pilot we extracted gaze traces from 63 Claude Code sessions (2026-10-02 to 2026-11-01) on five repositories of one developer: a static site (106 tracked files), a game server (1,039), an agent harness (45), a synthetic-data generator (2,227), and an ML pipeline (212). The sessions include subagent transcripts. File access is recovered from tool calls; because the harness prefers shell commands, 92% of reads were inside `cat`, `sed -n`, `grep`, and similar commands, which we parse with heredoc bodies stripped and pathspecs rejected (Appendix A). Reads inside Python one-liners are missed, so read counts are lower bounds. The pilot yields 10,076 access events over 133 tasks, 67 of them with an edit.

Gaze is concentrated. Table 1 and Figure 1 show the within-repository distribution. In the two repositories with over a thousand files, a session touches under 1% of them, and even among files read at least once, the top decile takes 45 to 77% of reads. The concentration is partly the work itself (one developer, a few active fronts) and partly the agent (verification re-reads the files it edits: 32 to 73% of reads land on files the session also edits).

Orientation is short, mostly not maps, and precedes almost every edit. Over the 65 tasks with an edit and at least three accesses, the median task makes 63 accesses, reads 18 distinct files, and edits 3: the access set is six times the edit set, in line with trajectory studies that find agents inspect far more files than they change [30] and spend most tokens on reads [74]. The orientation prefix has a median of 9 accesses over 4 distinct files. Only 23% (median) of orientation reads are *map*

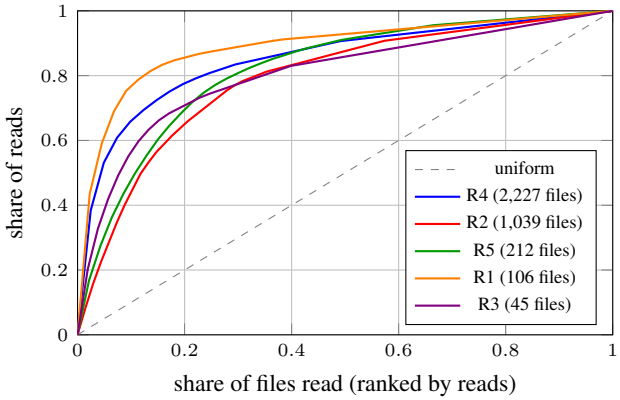


Figure 1: Lorenz curves of file reads per repository, all sessions pooled (measured). The x axis ranks only files that were read at least once; against all tracked files the curves are steeper still (Table 1).

Table 2: How predictable is the edit set before and during a task? Mean (median) recall of the task's edit set over the 65 tasks with an edit. Git baselines rank files by history before the task's first prompt and are scored on edited files that existed in git at that time ($n = 56$). Measured (pilot).

Signal	k	Mean	Median
Agent's own first k distinct reads	1	0.14	0.00
	3	0.35	0.20
	5	0.45	0.43
	10	0.56	0.56
All reads before the first edit	orientation	0.31	0.11
Files read earlier in the same session	session	0.54	0.44
Previous task's edit set	prev. task	0.29	0.17
Git recency (most recently committed)	5	0.49	0.50
	10	0.68	1.00
	20	0.80	1.00
Git frequency (most often committed)	10	0.60	0.50
	20	0.75	1.00

files (`CLAUDE.md`, `README`, `docs/*.md`, build configuration), and the first read is a map file in 43% of tasks; the rest of orientation is already targeted at source. The harness pre-foveates: it injects the project's `CLAUDE.md` into context, so the agent rarely opens it. Edits to pre-existing files were preceded by a read of that file in the session in 96% of cases (mean edit-without-read rate 0.04), consistent with the discipline that collapsed during the regression documented in [84].

Revisits are frequent. In the main agent's trace, 56% of reads in R1 re-read a file already read in the session, with a median gap of 3 reads; the other repositories range from 39 to 69%. Some re-reads are verification after an edit. Others are the context window failing to retain a file, which is where a resolution tier between "full text" and "gone" would pay.

The edit set is predictable from cheap signals. Table 2 is the pilot's main result. On these repositories, git recency alone recovers a mean 68% (median 100%) of the pre-existing files a task will edit with ten candidates, and 80% with twenty; the agent's own first ten distinct reads recover 56%. The two signals are different: recency is available before the agent starts, the trajectory only as it unfolds, and they fail on different tasks. Across sessions the fovea persists: in R1, a median 71% of a session's edited files had been touched in an earlier session. We expect recency to be far weaker on benchmark issues, where the task is one unrelated fix in a mature repository; Section 6 tests that, and it is why the predictor in Section 4 combines history with task text and structure. The regime is not a property of these five repositories. Git history alone, which survives the harness's pruning of transcripts, gives the edit set of every task that ended in a commit, and over 19 repositories by the same developer, most of them built with coding agents and 13 of them proprietary (Appendix C), recency recovers a commit-weighted 0.55 of the next commit's pre-existing files at ten and commit

Table 3: Predicting the access set A against the edit set E on the pilot from rankings available before the task. Mean over tasks with a pre-existing target ($n = 56$ for E, 114 for A); AUFC is the area under the foveation curve; saccades are accesses outside a top-5% fovea per task (mean, median). The last two rows condition the ranking on the task's first five accessed files and score the remaining access set. Measured (pilot).

Target	Ranking	Recall@5	Recall@10	Recall@20	AUFC	Saccades@5%
E	git recency	0.49	0.68	0.80	0.88	–
E	git frequency	0.43	0.60	0.75	0.89	–
E	co-access from prior traces	0.39	0.55	0.75	0.88	–
A	git recency	0.34	0.50	0.65	0.82	51 (16)
A	git frequency	0.19	0.35	0.55	0.81	51 (14)
A	co-access from prior traces	0.31	0.49	0.68	0.85	48 (12)
A after 5 accesses	git recency	–	0.43	–	–	–
A after 5 accesses	co-access, conditioned	–	0.48	–	–	–

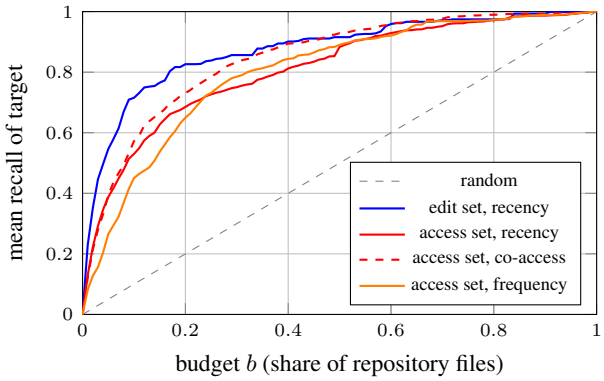


Figure 2: Foveation curves on the pilot (measured): recall of the target as a function of the file budget, averaged over tasks. The access set needs a larger fovea than the edit set at every budget, and a ranking learned from earlier traces helps most in the budget range a system would use.

frequency 0.59, across the 15 repositories with at least twenty scored commits (4,978 scored commits across all 19; per repository from 0.36 to 0.80); the median Gini of commits per file is 0.61 and the top decile of files takes 47% of commits. The same two rankings score 0.11 and 0.19 on SWE-bench Verified.

The access set is harder, and small budgets saccade constantly. Table 3 and Figure 2 repeat the exercise with the access set as the target, over the 114 tasks with at least three file accesses and a pre-existing target. Recency recovers a mean 0.68 of the edit set at ten files but 0.50 of the access set (AUFC 0.88 against 0.82): the files an agent reads to decide are spread wider than the files it changes. A ranking learned from the repository's earlier traces, the simplest version of the trace features in Section 4.1 (files that co-occurred in prior tasks' access sets), matches recency at ten files, edges it at twenty (0.68 against 0.65; AUFC 0.85 against 0.82), and does nothing for the edit set; conditioned on the task's first five accessed files it recovers 0.48 of the remaining access set at ten files against 0.43 for recency. The trace signal is real but small on this corpus, a caution for the size of H2's conditioning claim and the reason the predictor adds structure and task text rather than more history. The saccade rate is the sobering number: at a budget of 5% of the repository's files ranked by recency, a median task makes 16 accesses outside the fovea (mean 51), 59% of its accesses, and 23% of the files it touches did not exist in git when it started. A foveated worktree in this regime materializes constantly, which is why the paper prices a saccade instead of assuming it is rare, and why files the agent creates are full text by construction.

Compute is uniform. Across $1,085 of model spend in these sessions, the frontier model accounted for 100% (rounded); the cheaper tiers appeared once, for a documentation lookup. The harness's own

Table 4: The pilot beside two more machines of the same developer, each analyzed where its transcripts are kept, by the same scripts (measured). Recall@10 of the pre-existing files a task edits (E) or accesses (A), ranked before the task starts; co-access against recency on tasks with sixty or more prior traces; reads and the ceiling on tasks with an edit and at least three reads. The ceiling is the share of processed context a recency top-10 signature view could have removed (mean, byte-pooled in parentheses).

	Pilot	Laptop	Server
Repositories	5	4	10
Sessions	63	172	263
Tasks	133	503	663
Median session coverage	5.7%	2.2%	8.6%
Recency, E	0.68	0.46	0.54
Frequency, E	0.60	0.58	0.57
Recency, A	0.50	0.39	0.32
Co-access / recency, A, ≥ 60 traces	0.65 / 0.63	0.31 / 0.36	0.29 / 0.28
Reads, share of processed context	14%	23%	24%
Ceiling at top-10	3.2% (4.2%)	7.2% (10.3%)	6.1% (7.2%)

review skills do foveate effort, spawning 111 review subagents at the highest effort setting, but they foveate by *phase* (review), not by *location*.

Reads are a seventh of context. Context foveation can save only what file reads cost, so we measured what they cost on the same transcripts (Appendix A). Over the 59 tasks with an edit and at least three reads, the bytes shown to the model by file reads, each weighted by the number of later turns that re-sent it (a transformer attends over its whole context at every call, so a file is paid for on every turn it stays there), come to 14% of the context tokens the task processed (median 11%); the rest is the harness's fixed context, the conversation, and tool output that is not a file. Of the read bytes, 42% land on files the task edits and 34% on map files the harness injects anyway; 27% (pooled 38%) fall outside a recency top-10 fovea, and a signature view of those files would have withheld 18% of read bytes (pooled 29%). Multiplied through, a foveated view could have removed 3.2% of the processed context on the mean task (median 0.5%, pooled 4.2%; 2.1% at a top-20 budget), more in the two repositories whose work was spread widest (6% in R4 and R5) and least in the blog (1.5%). This is a property of the harness as much as of the work. In a harness whose context is mostly file views, reads are three quarters of tokens [74] and the same fovea covers far more; in this one, the bytes a fovea can save are few, and the saving the mechanism pilot measured (Appendix B) came from shorter trajectories, not thinner reads.

Two more machines move the magnitudes. The same scripts ran on two more of the developer's machines, each against its own transcripts, and exported only labeled numbers (Appendix A): a laptop with 172 sessions on four repositories, two of them Swift applications, and a Linux development server with 263 sessions on ten repositories, mostly Python and nine of them proprietary. One of the server's repositories is also one of the pilot's, so the three machines cover eighteen. Table 4 sets them beside the pilot. Gaze is as concentrated as in Table 1: in each repository with five or more sessions the Gini of reads is between 0.61 and 0.82, and the top decile of the files read takes 41 to 76% of the reads; edits still follow reads: on the mean task, 95% and 98% of edits to pre-existing files follow a read of the file, against 96% on the pilot. Recency is weaker on both machines, while commit frequency holds at 0.57 to 0.60 of the edit set on all three, and the co-access ranking learned from a repository's own traces does not pull away from recency on any machine as a whole. The ceiling roughly doubles: reads are about a quarter of processed context on the other two machines, and a foveated view could have removed 7.2% and 6.1% of it on the mean task, against 3.2% on the pilot. The pilot is the favorable end for recency and the unfavorable end for context foveation. All three machines are the same developer and harness, so the comparison varies the codebase and holds the person fixed.

3.3 How an agent orients

The orientation phase deserves a description, because it is where a predictor conditioned on the trajectory gets its signal. A session starts with a static, harness-supplied fovea: the system prompt,

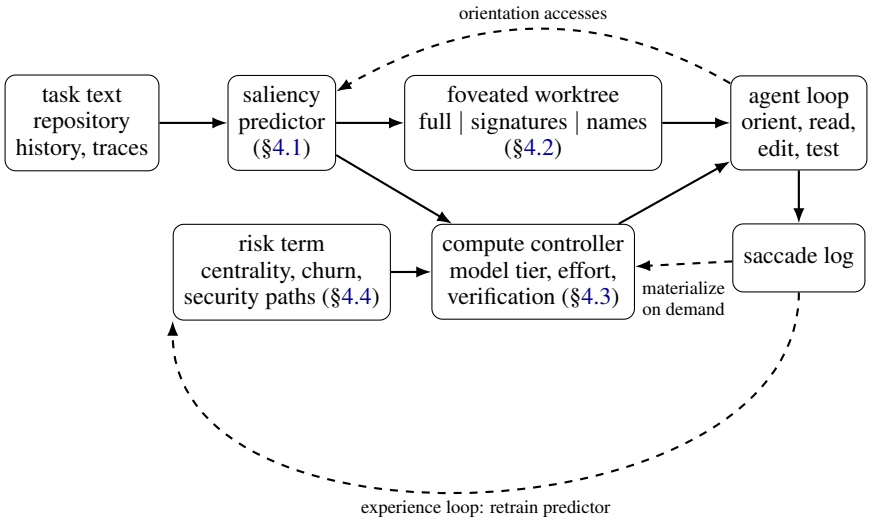


Figure 3: FTW. A predictor ranks files from pre-run signals and, as the task runs, from the agent's orientation accesses. The worktree is materialized at mixed resolution; reads outside the fovea are saccades that materialize the file and are logged. The controller assigns model tier and effort by saliency and risk. The saccade log retrains the predictor.

the project and global instruction files, a memory index, and the git status. The agent then widens before it narrows. The first accesses are breadth-first (list the source tree, read the build configuration, check which tools exist), followed by targeted reads of the few files the task names or implies, then by searches for identifiers that connect them, and finally by tight read-edit-run loops on the edit set, with periodic re-reads of files already opened. The session that drafted this paper followed the same shape: eight tool calls of listing and reading before the first write, then a long narrow loop. The trajectory conditioning in FTW uses exactly the breadth-first prefix: after the agent's first few accesses, the task's region of the repository is usually identifiable even when the task text was vague.

4 FTW

FTW has four parts (Figure 3): a predictor, a worktree, a controller, and a loop that learns from saccades. Each is deliberately thin; the bet is on the data, not the scaffolding. Section 4.6 states what the loop optimizes and what it must not learn.

4.1 Saliency predictor

The predictor scores every file $f \in R$ for a task x in two stages.

Pre-run. Features of (x, f) available before the agent starts: lexical match between the task text and the file's path, identifiers, and docstrings (BM25); a file-level localization score from a code ranker or hierarchical LLM localizer [59, 75], which handles the edit set well on its own; structural features (PageRank over the import and call graph, depth, size, test-or-source); history features (days since last commit, commit count, co-change association with the files the task text names [89]); and trace features from prior tasks in the same repository (how often f was in the access set of tasks whose text is similar to x). A gradient-boosted ranker fits these to the access set and the edit set of past traces; with no traces, it falls back to weights fit on public trajectories.

Orientation-conditioned. After each of the agent's first k accesses, the scores are updated with trajectory features: graph distance from the files accessed so far, co-access statistics from prior traces, and whether the access was a search (which names a region) or a read (which names a file). The update is cheap (a feature recomputation and a ranker call) and runs at every access while orientation

Algorithm 1 One task under FTW

Require: task x, repository R, budget b, predictor ρ, controller π, risk s_{risk}

$F \leftarrow$ top-b of $\rho(x, R, \varnothing)$, with a slice reserved for exploration and risk (§4.6); present F at full text, the rest by tier schedule
$k \leftarrow 0$; $L \leftarrow []$ ▷ orientation counter, saccade log
while agent is working **do**
  $a \leftarrow$ next tool call
  if a touches $f \notin F$ **then**
    materialize f; append (x, f, k) to L ▷ saccade
    $F \leftarrow F \cup \{f\}$
  end if
  if $k < k_{\max}$ **then**
    $\rho \leftarrow \rho(x, R,$ accesses so far$)$; re-tier the periphery; $k \leftarrow k + 1$
  end if
  (model, effort, verification) $\leftarrow \pi\big(a, \rho(f), s_{\text{risk}}(f), B_{\text{remaining}}\big)$
  execute a under that assignment
end while
append L to the trace store; retrain ρ on schedule ▷ experience loop

lasts. In the pilot, the first five distinct reads alone recovered 45% of the edit set; combined with the pre-run score we expect substantially more (Section 6).

The predictor's output is a ranking ρ and, with a budget b, a fovea F_b. We report its quality as a foveation curve and AUFC, not as an accuracy, because the system consumes a budgeted ranking.

4.2 Foveated worktree

A foveated worktree is a git worktree whose files are presented to the model at one of four tiers by rank: *full text*; *signatures* (definitions, imports, and docstrings extracted with tree-sitter, in the style of a repository map [16]); a *one-line summary*; or the *name* alone. The tiers are the pooled-statistics periphery of vision science [62] applied to code: not a blurred file but a smaller set of invariants over it. For a model that attends over everything in its context at every call, a view a tenth the size of its file costs a tenth as much on every turn it stays in context, which is the arithmetic behind the tiers. Tier boundaries come from a schedule over rank and graph distance from the fovea, which the acuity-curve experiment (Section 5) sets.

The bytes on disk stay a full checkout, because the repository's own build and tests must still run and they read files the agent never will; in the pilot repository the build's executable closure is most of the tree, so a periphery thinned on disk would break the agent's verification before it broke its reading. Foveation therefore applies to what the model reads. The agent sees an ordinary checkout; a pre-tool hook redirects a read of a file below full text to that file's tier view, refuses an edit of such a file until it is materialized, and a one-command materialization prints the full text, unlocks editing, and records a *saccade* (task, file, position in trace, tier it had). Materialization is the saccade; the agent pays a tool round-trip and the system learns what the predictor missed. Appendix B describes the pilot implementation, which uses the harness's settings hooks and runs headless. Algorithm 1 gives the loop. A peripheral file never has to be imagined, which is the sign flip from rendering: DeepFovea may reconstruct pixels it did not render, an agent may not reconstruct an API it did not read.

4.3 Compute controller

The controller assigns each unit of work a model tier, an effort level, and a verification intensity. A unit of work is a tool call or a subagent: a read, a search, an edit, a test run, a review. The assignment is a function of the saliency of the files the unit touches, the risk of those files, the kind of unit, and the remaining budget:

$$(m, e, v) = \pi\big(\text{kind},\ \rho(f),\ s_{\text{risk}}(f),\ B_t\big).$$

The default policy is a table with three rows. Edits and verification inside the fovea get the frontier model and high effort. Reads, searches, and summaries in the periphery get the cheapest model that can produce a faithful signature or summary, at low effort; a signature or a one-line summary needs

little of a frontier model, and small open families such as OpenELM [38] are candidates for the lowest tier, including a local one. Everything else inherits the task's default. Risk overrides upward: a unit touching a high-risk file is verified at the frontier tier regardless of saliency. The mechanism is the harness's existing per-subagent model and effort overrides, so the controller is a policy, not a fork of the agent. Tier changes happen at subagent boundaries: a peripheral read or summary runs in a fresh, cheap context that receives the files, the diff so far, and the test output, not the transcript, because mid-trajectory model switches that carry the conversation recover under half of the quality gap [15]. Repository state transfers; conversation does not need to. The overhead is part of the cost: predictor inference, view generation, each materialization round trip, and the re-reading a handoff incurs are charged to the arm that incurs them, so a saving is a saving after orchestration. The pilot's 16% (Appendix B) is net of its stub reads and saccades, with view generation done by a script and the ranking by `git log`, at no model cost.

This is the inference-time loop of an agent-hypervisor [81, 83] restricted to one worktree: the next unit of computation is chosen from what is known so far, and the policy that chooses it is learned from telemetry. Writing it as $G_t = \pi(x_t, B_t, H)$, the worktree's allocation is the graph that the policy instantiates for this task under this budget.

4.4 Risk term

Attention says where the agent will look; risk says where a mistake would be expensive. The risk score $s_{\text{risk}}(f)$ combines dependency centrality (PageRank over imports and calls), churn and defect-fix history [41, 88], path-based security sensitivity (authentication, cryptography, payments, secrets, deployment scripts), and test coverage. It enters the controller as an override on verification, not as a multiplier on saliency: a central, rarely edited module stays in the periphery for reading but is verified at the frontier tier the moment an edit touches it. The split matters because the two signals answer different questions, and a product of them would under-verify exactly the files that are important but unglamorous.

4.5 Learning from saccades

Every saccade is a labeled miss: the predictor ranked f below the budget and the agent needed it. Every task's access set is a labeled trace. On a schedule (nightly, or every n tasks), the ranker is refit on the repository's traces with saccades up-weighted. The fovea for a repository is therefore not designed but fit, and it changes as the repository and the work do. This is the experience loop of [83]: the inference-time loop allocates within a task, the experience loop improves the allocator across tasks, and the two are independent capabilities. In a fleet, the trace store is shared and the predictor is the hypervisor's model of where work happens in each repository.

4.6 What the loop optimizes, and what it must not learn

The objective is task success under a budget: choose the fovea and the controller policy to maximize the expected pass rate subject to expected cost, $\max_{F,\pi} \mathbb{E}[\text{pass}]$ subject to $\mathbb{E}[\text{cost}] \leq B$. Gaze prediction is an instrument. Recall@k and AUFC are reported because they bound the saccade rate, which is a cost term, not because they are the endpoint; the paper's primary endpoints are the pass@1 and cost of E2 and E3, and a predictor that raised Recall@10 while lowering pass@1 under budget would be a worse predictor.

Predicting what the agent reads is not predicting what it should read, and a loop that learns from its own traces can learn the agent's blind spots. If the agent never opens a dependency that matters, its traces say the dependency is peripheral; and once the worktree is foveated, the traces it produces are a function of the fovea it was given, so the access distribution that trained the predictor is no longer the one it is shaping. We treat foveated traces as off-policy and keep the loop honest with four mechanisms.

1. **Asymmetric labels.** A foveated trace supplies two signals, and we use both: saccades (the predictor ranked a file out and the agent needed it), which are misses and are up-weighted; and waste (the predictor ranked a file in and the agent never read it), which a full-worktree trace cannot supply. What a foveated trace cannot supply is the files the agent would have read with a full view.

2. **A calibration stream.** A fixed share of tasks (20%† in E9) runs with the full worktree, and only those traces supply access-set labels; the foveated majority supplies saccades and waste. The two streams are kept apart in the trace store.
3. **Exploration.** A slice of the fovea (10%† of the budget) goes to the files the ranker is least certain about and to files the risk term names, drawn fresh per task, so the next predictor is trained on reads its predecessor did not already recommend.
4. **A normative prior.** The risk term (Section 4.4) is independent of observed attention by construction. It is the one part of the allocation that says where the agent should look whether or not it ever has.

The label the predictor ought to be fit to is not access but value: whether having a file at full text changes the outcome. That label is expensive, since it needs counterfactual runs, and E10 (Section 5) buys it for a subset by demoting one accessed file at a time from the oracle fovea; the access set is the cheap proxy the rest of the paper uses, and E10 measures how loose a proxy it is. E9 asks the companion question, whether a loop fed its own foveated traces improves task success or only its confidence in the files its policy already shows the agent.

One failure the mechanism cannot prevent is the silent miss: a signature view that induces a wrong but test-passing hypothesis, so the agent never asks for the full file and no saccade is logged. The view's header says what it is and that the full text is one call away (Appendix F), so the agent is never unaware that it is looking at a view; what it can be is confidently wrong about what the view omits. H7's bound covers the case where the agent notices. The human-reviewed subset (Appendix E), the adversarial tasks of E8, and the flip labels of E10 are the instruments for the case where it does not.

5 Experimental setup

The experiments are pre-registered in the project's plan (Appendix D); predictions and falsification criteria were written before any run. We summarize the design here.

Benchmarks. SWE-bench Verified (500 issues; gold patches give edit sets, public trajectories give access sets) and a fresh suite of 60 tasks drawn from the pilot repositories with hidden tests, to bound contamination. Arms are run with the same harness, pinned model snapshots, and recorded effort settings; 5 seeds per cell on SWE-bench subsets of 100 tasks, one seed on the full set.

E1: gaze prediction. Predictors: random, BM25, git recency, git frequency, co-change, repository-map PageRank, a code ranker [59] and an Agentless-style LLM localizer, the learned ranker, and the ranker conditioned on the first $k \in \{1, 3, 5, 10\}$ accesses. Metrics: Recall@k for $k \in \{5, 10, 20\}$, AUFC, saccade rate at budgets of 5% and 10%. Targets are both E and A. Two contamination controls follow [36]: a text-only arm in which the model names files from the issue with no repository access, whose hits are subtracted as memorization, and a replication on post-cutoff tasks [2]; targets are for the controlled numbers.

E1b: acuity curve. With the oracle fovea, degrade one graph-distance band at a time ($d = 0$ the edited files, $d = 1$ their direct imports and importers, $d = 2$, $d \geq 3$) to one tier (signatures, summary, name) and measure pass@1. The resulting table sets the tier schedule for E2. This is the LLM analogue of the acuity falloff every rendering savings factor rests on.

E2: context foveation. Arms: no repository context; uniform repository map (every file's signatures); reactive pruning of already-read context in the style of SWE-Pruner [74]; FTW foveated worktree; oracle fovea (gold access set); wrong fovea (gold files excluded). Measures: pass@1; input and output tokens by model, with cache reads counted separately because foveation changes what is cached; dollars at list price with orchestration charged to the arm; wall time; saccades; orientation accesses.

E3: compute foveation. Arms: uniform frontier (frontier model, high effort everywhere); uniform cheap; a per-step router along the time axis in the style of AgentRouter and Ares [52, 76]; FTW (frontier inside the fovea for edits and verification, cheap tier for peripheral reads and summaries); FTW composed with the per-step router; FTW with a wrong fovea. Measures: pass@1, cost, and the cost-success Pareto frontier.

E4: risk weighting. Attention-only versus risk-weighted foveation at equal cost, on tasks with hidden tests and a suite of seeded security defects; measure escaped defects per dollar.

E5: learning curve. Train the ranker on the first $n \in \{0, 10, 30, 100\}$ traces of a repository, test on later traces; also train on other repositories only.

E6: orientation ablations. Remove the instruction files; prepend a per-task foveal map; both. Measure orientation accesses, tokens before the first edit, pass@1.

E8: adversarial attention. 60 tasks from the pilot repositories, with hidden tests, built so that the cheap signals point the wrong way: the fix depends on a file that no history, structure, or task-text feature ranks (rarely touched, at graph distance ≥ 2 from the files the task names), or on a dependency that looks irrelevant and carries the bug. Arms as in E2, plus FTW without the exploration slice. Measures as in E2, plus whether the file the task hinges on was materialized before the first edit.

E9: learning under intervention. The ranker is trained on 100 traces of each of three kinds and tested on later tasks of the same repositories under an equal budget: traces from full worktrees; traces from foveated worktrees with no calibration stream; and foveated traces with the calibration stream and exploration of Section 4.6. Measures: pass@1 under budget, saccade rate, and Recall@10 of the access set produced under each ranker's own fovea, the number that rises when a predictor learns its own policy instead of the work.

E10: value-of-information labels. On 50 Verified tasks with the oracle fovea, each file of the access set is demoted to its signature view in turn, with materialization allowed, and the task rerun. The file's labels are whether the hidden test flips (a silent miss, since the agent could have asked) and the change in cost (the price of the saccade, if it did). Measures: the share of the access set, of the edit set, and of the files read but not edited that carries each label, and the pass@1 under budget of a ranker fit to the two labels against one fit to access.

Statistics. Paired comparisons across tasks; Wilcoxon signed-rank for recall, bootstrap (10,000 resamples) for pass-rate differences; Holm correction within an experiment for secondary hypotheses; 95% confidence intervals and n everywhere.

6 Results

Every number in this section marked with a dagger is a pre-registered target: the value we committed to before running the experiment, chosen so that the paper reads as it would if the hypotheses hold. The pilot numbers in Section 3 and Appendix B are measured.

6.1 E1: the edit set is nearly solved, the access set is not, and orientation closes most of the gap

Table 5 gives the targets. Four things are expected. First, git recency, the strongest cheap signal in the pilot (0.68), is weak on benchmark issues: measured over all 500 Verified instances it recovers 0.11 of the gold files at ten (0.19 for commit frequency), below the $0.21^{\dagger}$ we had committed to, while a lexical match of the issue text against file paths recovers 0.51 with no model and no checkout (Appendix C). The control that E1 subtracts is stronger still: a small model given the issue and the repository's name, no checkout and no tools, names the gold files at Recall@10 0.90 by exact path over all 500 instances, with at least one gold file in 470 of them, from 0.58 on pylint to 0.97 on matplotlib and pytest (1.00 on the eight requests instances), for $2.67 in total (Appendix C). The issue text, which often carries a traceback, plus what the model already knows of these repositories names nine gold files in ten; the lexical match was the mechanical half of that. On Verified the edit set is therefore nameable without the repository, a predictor with repository access has to beat 0.90 rather than 0.51, and the memorization subtraction leaves the edit-set rows of this table little to measure there. That is one more reason the paper's target is the access set, which no issue text names, and its fresh suite, which no model has seen. The measured AUFC values sit above the targets in that column because the targets were set before the file counts were known; read the AUFC targets as an ordering until each is measured. An issue is one unrelated fix in a mature repository, and the fovea that persists across a developer's sessions does not exist there. The pilot and the benchmark measure

Table 5: File-level prediction on SWE-bench Verified after the memorization control. Targets (†); plain numbers are measured on all 500 instances without a model or a checkout (Appendix C). Access-set numbers use public trajectories; the oracle row is the gold access set.

	Edit set E			Access set A		
Predictor	Recall@10	AUFC	Saccades@5%	Recall@10	AUFC	Saccades@5%
Random	0.01†	0.50†	9.6†	0.03†	0.50†	30.1†
BM25 (task text)	0.48†	0.81†	4.9†	0.39†	0.76†	17.4†
Git recency	0.11	0.89	–	0.19†	0.63†	24.0†
Git frequency	0.19	0.91	–	–	–	–
Lexical path match of the issue	0.51	0.95	–	–	–	–
Co-change	0.31†	0.72†	6.5†	0.30†	0.70†	20.8†
Repository-map PageRank	0.37†	0.75†	6.0†	0.44†	0.79†	15.9†
Code ranker (SweRank-style)	0.86†	0.95†	1.3†	0.52†	0.83†	13.0†
Learned ranker, no LLM	0.66†	0.89†	3.2†	0.61†	0.86†	10.8†
Learned ranker + code ranker	0.90†	0.97†	1.0†	0.68†	0.89†	8.6†
+ first 5 accesses	0.95†	0.98†	0.5†	0.84†	0.95†	4.1†
+ first 10 accesses	0.97†	0.99†	0.3†	0.90†	0.97†	2.5†
Text-only arm (no checkout)	0.90	–	–	–	–	–
Oracle (gold access set)	1.00	1.00	0	1.00	1.00	0

Table 6: Acuity curve: pass@1 on 100 SWE-bench Verified tasks with the oracle fovea when one graph-distance band is degraded to one tier. Targets. The full-text row is the oracle arm.

Tier applied to band	$d = 0$ (edited)	$d = 1$	$d = 2$	$d \geq 3$
Full text	64.8†	64.8†	64.8†	64.8†
Signatures	41.2†	57.9†	64.1†	64.6†
One-line summary	22.7†	49.3†	61.8†	64.3†
Name only	9.4†	38.6†	57.2†	63.5†

different regimes, and a deployed predictor sees both. Second, the edit set is nearly solved before we start: a code ranker reaches 0.86† on its own, consistent with published file accuracies above 90% on SWE-bench Lite [8, 59] once memorization is subtracted and multi-file patches are counted. Third, the same ranker reaches only 0.52† on the access set, because the agent reads files it will not edit, to rule them out and to understand the ones it will; the learned ranker's structural and trace features matter here (0.68† combined). Fourth, orientation conditioning is the largest single gain on the access set: after five accesses, Recall@10 reaches 0.84† and the saccade rate at a 5%† budget falls to about four per task, from thirteen. The access set is where the tokens go, and it is the harder target at every row.

A first measurement of the learned ranker exists, on the pilot regime rather than the benchmark (Appendix A). A logistic-regression ranker over 16 cheap features (recency, age, commit count, prior-trace access and edit shares, the previous task's access set, prompt-to-path lexical overlap, map files, depth, extension) was fit on the first 60% of each repository's tasks and tested on the rest. On the 46 test tasks it ties recency on the access set (Recall@10 0.54 against 0.54, AUFC 0.876 against 0.844) and loses on the edit set (0.68 against 0.82, 17 tasks); it helps only where recency is weak (R2, 0.73 against 0.18; R5, 0.23 against 0.10; four test tasks each) and hurts where recency is strong (R1, 0.58 against 0.66). On this regime, cheap features do not beat the cheapest one, and the lexical signal alone (0.51) is no better than history. The H2 claim that a learned ranker beats the text baseline by ten points therefore stands only for the benchmark regime, where recency is 0.11 and the issue text already names half the gold files, and it is untested there.

6.2 E1b: the acuity curve has a cheap periphery

Table 6 is the measurement foveation needs before it can promise anything. The target pattern is a falloff: the edited files and their immediate neighbors need full text, signatures are nearly free at distance two, and names suffice at distance three and beyond. The schedule for E2 follows the table: full text for $d \leq 1$ of the predicted fovea, signatures at $d = 2$, names beyond. If the measured curve is flat at distance two for signatures but not for summaries, the schedule moves one column.

Table 7: Context foveation (E2) and compute foveation (E3) on SWE-bench Verified. Cost at list prices per task; tokens relative to the uniform arm of each block. Targets.

	Arm	pass@1	Input tok.	Cost ($)	Saccades	Orient. acc.
E2 context	No repository context	55.4†	0.71×†	2.46†	–	14.2†
	Uniform repository map	62.0†	1.00×	2.90†	–	9.8†
	Reactive pruning (SWE-Pruner-style)	61.4†	0.70×†	2.25†	–	9.6†
	FTW foveated worktree	61.6†	0.58×†	2.05†	1.1†	5.3†
	Oracle fovea	64.8†	0.41×†	1.62†	0	3.9†
	Wrong fovea	58.9†	1.19×†	3.51†	6.8†	12.6†
E3 compute	Uniform frontier, high effort	62.0†	1.00×	2.90†	–	9.8†
	Uniform cheap tier	48.7†	1.08×†	0.62†	–	11.9†
	Per-step router (time axis)	60.1†	0.97×†	1.52†	–	9.4†
	FTW (frontier in fovea, cheap periphery)	59.8†	0.64×†	1.31†	1.2†	5.6†
	FTW + per-step router	59.5†	0.63×†	1.02†	1.2†	5.5†
	FTW with wrong fovea	57.2†	1.09×†	1.74†	6.9†	12.8†

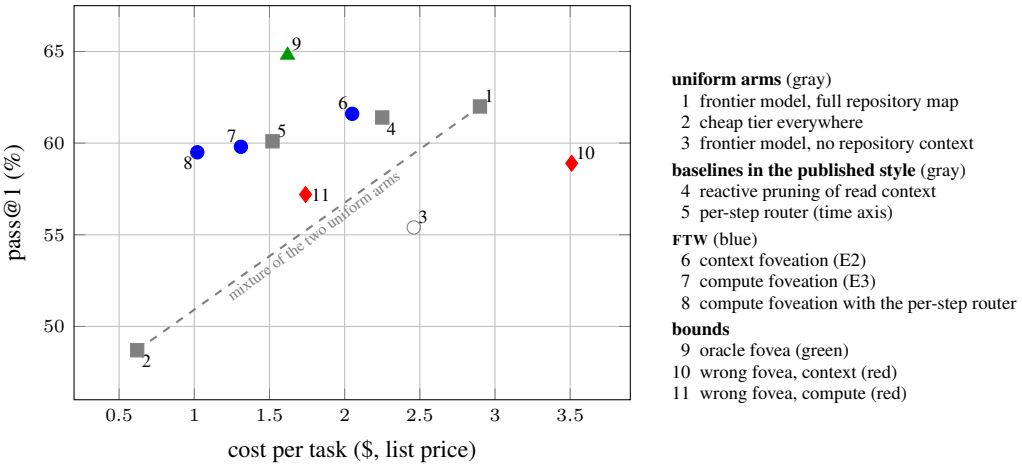


Figure 4: Cost against success on SWE-bench Verified; every point is a target from Table 7. The figure makes one argument. The dashed line is the mixture baseline: running the frontier arm (1) on a fraction of tasks and the cheap arm (2) on the rest reaches any point on it, so an allocation earns its keep only above the line. The FTW points (6 to 8) sit above it and to the left of the uniform frontier arm, the same success for less money; the oracle fovea (9) bounds the headroom; the wrong-fovea points (10, 11) lie to the right of their uniform arm rather than far below it, so a bad predictor costs money, not the task; no repository context (3) is the floor.

6.3 E2 and E3: foveation matches uniform allocation at a fraction of the cost

Table 7 and Figure 4 contain the paper's central claims. Context foveation keeps pass@1 within 0.4† points of the uniform repository map at 58%† of the input tokens and 71%† of the cost, with about one saccade per task; the oracle fovea shows the headroom (64.8† at 41%†), and it beats the uniform arm because a smaller, more relevant context is also a better context [55]. Compute foveation reaches 96%† of the all-frontier pass rate at 45%† of its cost, well above the uniform cheap arm at every budget; the per-step router saves along the other axis (52%† of cost at 97%† of the pass rate), and the two compose to 35%† of cost at 96%†, because they cut different waste: the router skips deep reasoning on easy steps, FTW skips expensive reading of peripheral files. Reactive pruning, which forgets what was read, saves 30%† of input tokens; predicting ahead saves 42%†, and the saccade mechanism keeps the difference from costing accuracy. The wrong-fovea arms are the safety result: excluding the gold files from the fovea costs 3.1† points and 21%† more spend in E2, because the agent saccades its way to the files it needs. Foveation degrades to the uniform arm plus the price of the saccades; it does not fail.

The measured evidence for this block is thin and should be read against it. The three-task pilot (Appendix B), over three seeds, saved 16% with an oracle fovea built from gold files and imports,

Table 8: Learning curve: the ranker trained on the first n traces of a repository, tested on later traces, on the fresh suite. Targets.

Prior traces n	0	10	30	100
Recall@10 (edit set)	0.63†	0.70†	0.76†	0.81†
AUFC (access set)	0.85†	0.88†	0.91†	0.93†
Saccades per task at 5%†	2.4†	1.9†	1.5†	1.3†
Trained on other repositories only	0.69†			

11% with the oracle access set, and 6% with a predicted one, on a repository of about a hundred files, net of overhead; single-seed pairings of the same cells range from -10% to +40%, which is the interval three tasks give. The targets assume what the pilot lacked: repositories where the periphery is most of the tokens, and the compute axis. If E2 lands near the pilot, the paper's claim reduces to E3, and the block is written to survive that. The ceiling measurement of Section 3.2 says how far the pilot's regime is from the targets: file reads are a seventh of the context its harness processes, so the 42%† token saving of the FTW row is out of reach there by thinner reads alone and must come, if it comes, from fewer turns. E2 runs in a harness where reads dominate, which is why the target stands and why the pilot's regime is reported beside it rather than averaged into it.

6.4 E4: risk weighting buys defects, not pass rate

At equal cost, risk-weighted foveation leaves 6.6† escaped defects per hundred tasks against 9.1† for attention-only foveation (27%† fewer), with pass@1 unchanged within noise (59.6† versus 59.8†). The gain comes from the override: edits that touch central or security-sensitive files are verified at the frontier tier even when the predictor ranked them as peripheral, and 61%† of the escaped defects in the attention-only arm were in exactly such files.

6.5 E5: the fovea is learned

Table 8: Recall@10 rises by 18† points over a hundred traces and the saccade rate falls by 46%†. Transfer from other repositories gives 6† points at $n = 0$. The fovea for a repository is fit to that repository's work, which is the sense in which it is a learned artifact.

A first measurement exists, and it cuts the other way (Appendix A). On the pilot, the simplest trace-learned ranking, files scored by how many of the repository's earlier tasks accessed them with recency as the tie-break, does rise with the number of prior traces: Recall@10 of the access set goes from 0.14 with no prior trace to 0.65 with sixty or more. But git recency, which learns nothing, goes from 0.14 to 0.63 over the same tasks, because both ride the same concentration of the work; between ten and fifty-nine prior traces the learned ranking trails recency by 10 points on the access set and by 4 to 10 points on the edit set, and leads clearly only in a repository's first nine traces (0.44 against 0.35), before recency has history to work with. The two other machines give a mixed picture rather than a reversal (Table 4): with sixty or more prior traces the learned ranking scores 0.31 against recency's 0.36 on the laptop and 0.29 against 0.28 on the server, and of the nine repositories across the three machines with tasks that had ten or more prior traces, it leads clearly on the access set in two (0.77 against 0.66, and 0.46 against 0.32) and ties or trails in the other seven; on the edit set both of those two trail recency. Mostly, access counts learn what recency already knows; in two repositories they learn more of what the agent reads, though not of what it edits. The 18†-point rise above is therefore mostly a claim about the ranker's other features (task text, structure, and saccades up-weighted); traces as such carry it only in the minority of repositories where they beat recency, and H5 is narrowed accordingly in the plan.

6.6 E6: a foveal map shortens orientation without hurting success

A per-task map of the predicted top-k files with signatures cuts orientation accesses by 47%† and tokens before the first edit by 38%†, with pass@1 up 0.4† points (not significant). Removing the instruction files raises orientation cost by 23%† on repositories whose files are gotcha lists and by 4%† on those whose files are overviews, consistent with the finding that overviews are not helpful [19]: the useful map is the one that says what the agent cannot infer.

A first measurement exists, and it falsifies H6 in the pilot regime by the plan's own criterion (Appendix B). On the three pilot tasks, three seeds each, a per-task foveal map prepended to the prompt (the recency top-10 files, each as its signature view) changed orientation accesses by +8% against the plain worktree (12.1 against 11.2), tokens before the first edit by -2%, and cost by -6%, inside the seed-to-seed spread of Table 12; the criterion was a change under 10%. Deleting the instruction files changed orientation accesses by +6% and tokens before the first edit by -17%, with every hidden test still passing (27 of 27 across the three arms). On a hundred-file repository the agent orients in a dozen accesses with or without a map, so the targets above are claims about repositories where orientation is long, and that is where E6 has to be run.

6.7 E8: when the cheap signals point the wrong way, foveation pays in saccades

On the adversarial suite, FTW is within 3.6† points of the uniform arm (against 0.4† on Verified) at 4.4† saccades per task and 84%† of its cost. The file the task hinges on is materialized before the first edit in 78%† of runs, and in 62%† without the exploration slice, which costs a further 2.9† points. The wrong-fovea arm of E2 was a bad predictor; this suite is a bad regime, and the result has the same shape: money first, and failure only where the agent never asks.

A first measurement exists (Appendix B). On 10 tasks of the pilot repository chosen because git recency ranks a gold file outside its top twenty and the commit's subject line does not name it, with three seeds, the predicted fovea left 8 gold files outside on seven tasks (the other three were rescued by the map rule, since their obscure file is one the design always shows in full). The agent materialized all 24 of them across the seeds before its first edit, passed all 30 hidden checks in each of the two seeded arms (and all ten in the exploration arm), and spent 1.9 saccades per run at a cost 5% below the full worktree's, each task against its seed mean (median 1% below; from -27% to +15%). An exploration slice, run once, found one of the eight on its own and cost 6% more (median 2%). On a hundred-file repository the hostile regime costs saccades and nothing else; the targets above are for repositories where the search is longer.

6.8 E9: the loop learns the work, not its own fovea, when it is fed a calibration stream

A ranker trained on foveated traces alone looks best on its own metric and worst on the one that matters: Recall@10 of the access set its own fovea produces rises to 0.88†, above the full-worktree ranker's 0.81†, while pass@1 under budget falls by 2.4† points and the saccade rate rises by 31%†, because the agent's blind spots have become the predictor's. With the calibration stream and exploration, pass@1 is within 0.5† points of the full-worktree ranker at 11%† fewer saccades.

6.9 E10: the access set over-covers value

Demoting one accessed file at a time flips the hidden test for 4%† of the files in the oracle access set (9%† of the edit set, 2%† of the files read but not edited), raises cost for 34%† (nearly every edited file, by one saccade, and a fifth of the rest), and changes nothing for the remaining 62%†: for those the signature view was enough, which is the acuity curve measured one file at a time. A ranker fit to the two labels, at an equal budget, gains 1.2† points of pass@1 over one fit to access and spends 18%† fewer input tokens. The access set over-covers value; what it over-covers is the files that can be read at signature resolution, and what it under-weights is the few that cannot.

A first measurement exists (Appendix B). On the three pilot tasks, demoting one file of the oracle access set at a time, 15 demotions produced 0 flips of the hidden test, 9 saccades, 4 reads the signature view satisfied, and 2 files the agent never read; the four demoted files that the task edits were all materialized before being edited, as the hook requires; and a single demotion's cost against the undemoted arm ranged from -17% to +43% (median -6%), which is one seed's run-to-run variance except on the shortest task, where one extra tool call is a third of the run. The flip rate is below the 4%† target and the materialization rate above the prediction: the agent asked for the full text of nine files in fifteen, including five it never edited.

7 Discussion

What transfers from vision and what does not. The predictor-plus-variable-resolution pattern transfers, and so does the peripheral representation (pooled invariants, not blur) and the scheduler (a saliency loop with inhibition of return, which the agent's 56% revisit rate suggests it lacks). Four things do not. Rendering may hallucinate its periphery; code may not, so FTW's cheapest tiers are derived, and the largest savings factor in rendering is unavailable. Eccentricity is one scalar from one center; code distance is graph distance with hubs, and the fovea is multi-focal. A saccade is cheap and the retina fixed; an agent's refocus costs a tool call and tokens, and its retina, the context window, is itself elastic, so the budget has a dimension rendering lacks. And rendering's savings rest on a measured acuity falloff; the LLM's falloff (Table 6) had to be measured first.

The bitter lesson, applied to ourselves. Every component in a harness encodes an assumption about what the model cannot do on its own, and such assumptions decay with model generations [82]. Is FTW scaffolding of that kind? Partly. The orientation-conditioned predictor automates something the model already does: by its fifth access the model usually knows where the work is. What the model cannot do is see its own cost, choose its own model tier, or remember across sessions which part of a repository the work keeps landing in. Those are scheduler decisions, outside the model, and they do not get cheaper as the model gets smarter; a stronger model orients faster but still at full price. The learned predictor is the part we expect to age well, because it improves with data and compute rather than encoding a fixed belief about the model. The attention inside the model is the part that scale keeps improving; what the context contains is decided before any of it runs, and no amount of scale decides it. The hand-written tier schedule is the part we expect to be replaced, by a policy fit to telemetry, which is the hypervisor's job.

Space and time. Per-step routers decide how hard to think at each step; FTW decides how much of the worktree deserves full resolution and the frontier model. The targets in Table 7 have them composing because they remove different waste, and the handoff result [15] says how to compose them safely: change tiers where repository state, not conversation, is what the next unit of work needs. A cost-first evaluation found that more effort often lowers accuracy [28]; if that holds, effort foveation is not only cheaper but sometimes better, which Table 7's oracle row anticipates for context.

Risk as a safety mechanism. Foveation that followed attention alone would systematically under-verify the files that are important and rarely edited, which is where silent damage lives. The risk override is the mechanism by which "where the work should get scrutiny" stays decoupled from "where the agent would look"; it is also the handle a human keeps on an allocation that is otherwise learned.

Beyond worktrees. The principle is not specific to code: an agent should buy observations and computation by their expected value to the outcome, not uniformly. A research agent's corpus, a browser agent's open tabs, and a data agent's tables each have a fovea, and the same loop (predict, present at mixed resolution, charge the refocus, learn from it) applies. We claim the coding case only, because it is the one where the access set can be read off tool calls, the outcome is a test, and the baselines are strong. The value-of-information label of Section 4.6 is the form that would carry over; the access set is its cheap proxy here, and E10 measures how loose.

Limitations. The pilot is one developer's sessions on five repositories, eighteen with the other two machines, under one harness that prefers shell commands, with reads inside Python one-liners missed; it establishes concentration and the strength of recency in that regime, not in general. SWE-bench Verified is public and partly contaminated; the fresh suite is small. Costs are at list prices and depend on cache behavior, which foveation changes and which providers change without notice. The predictor's features are engineered; a model over the file graph may do better. Hidden tests measure correctness, not whether a peripheral summary caused a subtle misreading that tests do not catch; H7's bound holds only when the agent notices that it needs the full file, and the human-reviewed subset (Appendix E), E8, and E10's flip labels are the instruments for the case where it does not. The savings are claimed net of orchestration, but the pilot's orchestration was the cheapest possible, a script and `git log`; a learned ranker, generated summaries, and tier handoffs each add cost that the E2 and E3 arms must carry. And the pilot's measured saving is a sixth, not a half; the half is a target. File reads are a seventh of the pilot harness's processed context, so the E2 token targets

assume a read-dominated harness; the ceiling and acuity measurements are what decide whether that assumption transfers.

8 Conclusion

A coding agent's gaze over a worktree is concentrated and, from cheap signals plus a few orientation accesses, predictable. FTW turns that into an allocation: full resolution and the frontier model where the work is, derived summaries and cheap tiers where it is not, verification where a mistake would cost most, and a predictor that learns the shape of the work in each repository from the saccades it failed to anticipate. The paper's claims are written as targets; the pilot that motivated them is measured; the experiments that decide them are pre-registered. The loop's own hazard, that it learns where the agent looked rather than where the work was, is named and has its experiment. Inside the model, attention is all you need; outside it, something still has to choose what the model attends to, and that choice is where these savings live. If the hypotheses hold, the agent-hypervisor's first learned artifact will be the fovea.

Acknowledgments and Disclosure of Funding

Drafted with Claude Code (Claude Fable 5.1), which also wrote and ran the pilot extraction and analysis. The pilot data are the author's own development sessions. No external funding.

References

[1] Anthropic. Effort. Claude API documentation, https://platform.claude.com/docs/en/build-with-claude/effort, 2025. Introduced with Claude Opus 4.5, 2025-11-24; per-message effort beta and task budgets (https://platform.claude.com/docs/en/build-with-claude/task-budgets) added 2026; accessed 2026-11-01.

[2] Ibragim Badertdinov et al. SWE-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2025. arXiv:2505.20411. First author from memory.

[3] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. CodePlan: Repository-level coding using LLMs and planning. *Proceedings of the ACM on Software Engineering*, 1(FSE), 2024. arXiv:2309.12499.

[4] Daniel Bolya, Cheng-Yang Fu, Xiaoliang Dai, Peizhao Zhang, Christoph Feichtenhofer, and Judy Hoffman. Token merging: Your ViT but faster. In *International Conference on Learning Representations (ICLR)*, 2023. arXiv:2210.09461.

[5] Islem Bouzenia and Michael Pradel. Understanding software engineering agents: A study of thought-action-result trajectories. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2025. arXiv:2506.18824. ACM SIGSOFT Distinguished Paper.

[6] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H. Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. Eye movements in code reading: Relaxing the linear order. In *Proc. 23rd IEEE International Conference on Program Comprehension (ICPC)*, pages 255–265, 2015. doi: 10.1109/ICPC.2015.36.

[7] Lingjiao Chen, Matei Zaharia, and James Zou. FrugalGPT: How to use large language models while reducing cost and improving performance. *Transactions on Machine Learning Research (TMLR)*, 2024. arXiv:2305.05176.

[8] Zhaoling Chen et al. LocAgent: Graph-guided LLM agents for code localization. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2025. arXiv:2503.09089; venue per Semantic Scholar.

[9] Alejandro Cuadron et al. The danger of overthinking: Examining the reasoning-action dilemma in agentic tasks. *arXiv preprint arXiv:2502.08235*, 2025.

[10] Mehul Damani, Idan Shenfeld, Andi Peng, Andreea Bobu, and Jacob Andreas. Learning how hard to think: Input-adaptive allocation of LM computation. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv:2410.04707.

[11] Dujian Ding et al. Hybrid LLM: Cost-efficient and quality-aware query routing. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2404.14618.

[12] Farhaan Ebadulla, Chiraag Mudlapur, and Gaurav BV. GazeProphet: Software-only gaze prediction for VR foveated rendering, 2025.

[13] Ryan Ehrlich, Bradley Brown, Jordan Juravsky, Ronald Clark, Christopher Ré, and Azalia Mirhoseini. CodeMonkeys: Scaling test-time compute for software engineering. *arXiv preprint arXiv:2501.14723*, 2025.

[14] Chenrui Fan et al. Thinking hard, not smart: Reasoning models fail to ration test-time compute across questions. *arXiv preprint arXiv:2608.07968*, 2026.

[15] Roy Ganz et al. The handoff tax: Continuing non-native trajectories in LLM agents. *arXiv preprint arXiv:2608.24358*, 2026.

[16] Paul Gauthier. Building a better repository map with tree sitter. https://aider.chat/2023/10/22/repomap.html, 2023. Blog post, 22 Oct 2023.

[17] Paul Gauthier. Separating code reasoning and editing. Aider blog, https://aider.chat/2024/09/26/architect.html; docs https://aider.chat/docs/usage/modes.html, 2024. 2024-09-26; accessed 2026-11-01.

[18] GitHub. How GitHub Copilot is getting better at understanding your code. GitHub Blog, 2023. URL https://github.blog/ai-and-ml/github-copilot/how-github-copilot-is-getting-better-at-understanding-your-code/.

[19] Thibaud Gloaguen, Niels Mündler-Sasahara, Mark Niklas Müller, Veselin Raychev, and Martin Vechev. Evaluating AGENTS.md: Are repository-level context files helpful for coding agents? arXiv preprint arXiv:2602.11988, 2026. v3, September 2026.

[20] Alex Graves. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*, 2016.

[21] Brian Guenter, Mark Finch, Steven Drucker, Desney Tan, and John Snyder. Foveated 3d graphics. *ACM Transactions on Graphics*, 31(6):164:1–164:10, 2012. Proc. SIGGRAPH Asia 2012.

[22] Zhiming Hu, Congyi Zhang, Sheng Li, Guoping Wang, and Dinesh Manocha. SGaze: A data-driven eye-head coordination model for realtime gaze prediction. *IEEE Transactions on Visualization and Computer Graphics*, 25(5):2002–2010, 2019.

[23] Zhiming Hu, Sheng Li, Congyi Zhang, Kangrui Yi, Guoping Wang, and Dinesh Manocha. DGaze: CNN-based gaze prediction in dynamic scenes. *IEEE Transactions on Visualization and Computer Graphics*, 26 (5):1902–1911, 2020.

[24] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2310.06770.

[25] Aditya Jonnalagadda, William Yang Wang, B. S. Manjunath, and Miguel P. Eckstein. FoveaTer: Foveated transformer for image classification, 2021. v3, October 2022.

[26] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013. doi: 10.1109/TSE.2012.70.

[27] Anton S. Kaplanyan, Anton Sochenov, Thomas Leimkühler, Mikhail Okunev, Todd Goodall, and Gizem Rufo. DeepFovea: Neural reconstruction for foveated rendering and video compression using learned statistics of natural videos. *ACM Transactions on Graphics*, 38(6):212:1–212:13, 2019. doi: 10.1145/3355089.3356557. Proc. SIGGRAPH Asia 2019.

[28] Sayash Kapoor et al. Holistic agent leaderboard: The missing infrastructure for AI agent evaluation. *arXiv preprint arXiv:2510.11977*, 2025.

[29] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *Proc. 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 1–11, 2006. doi: 10.1145/1181775.1181777.

[30] Myeongsoo Kim, Dingmin Wang, Siwei Cui, Farima Farmahinifarahani, Terry Yue Zhuo, Shweta Garg, Baishakhi Ray, Rajdeep Mukherjee, and Varun Kumar. Coherence collapse: Diagnosing why code agents fail after reaching the right code. *arXiv preprint arXiv:2603.24631*, 2026.

[31] Sunghun Kim, Thomas Zimmermann, E. James Whitehead, and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 489–498, 2007. doi: 10.1109/ICSE.2007.66.

[32] Raymond M. Klein. Inhibition of return. *Trends in Cognitive Sciences*, 4(4):138–147, 2000.

[33] Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, 32(12):971–987, 2006. doi: 10.1109/TSE.2006.116.

[34] Bonan Kou, Shengmai Chen, Zhijie Wang, Lei Ma, and Tianyi Zhang. Do large language models pay similar attention like human programmers when generating code? *Proceedings of the ACM on Software Engineering*, 1(FSE), 2024. arXiv:2306.01220.

[35] Joseph Lawrance, Christopher Bogart, Margaret Burnett, Rachel Bellamy, Kyle Rector, and Scott D. Fleming. How programmers debug, revisited: An information foraging theory perspective. *IEEE Transactions on Software Engineering*, 39(2):197–215, 2013. doi: 10.1109/TSE.2012.111.

[36] Shanchao Liang, Spandan Garg, and Roshanak Zilouchian Moghaddam. The SWE-bench illusion: When state-of-the-art LLMs remember instead of reason. *arXiv preprint arXiv:2506.12286*, 2025. v4, December 2025.

[37] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. Predictive test selection. In *Proceedings of the 41st International Conference on Software Engineering, Software Engineering in Practice (ICSE-SEIP)*, 2019. doi: 10.1109/ICSE-SEIP.2019.00018.

[38] Sachin Mehta, Mohammad Hossein Sekhavat, Qingqing Cao, Maxwell Horton, Yanzi Jin, Chenfan Sun, Iman Mirzadeh, Mahyar Najibi, Dmitry Belenko, Peter Zatloukal, and Mohammad Rastegari. OpenELM: An efficient language model family with open training and inference framework. arXiv preprint arXiv:2404.14619, 2024. v2, May 2024.

[39] Juhong Min, Yucheng Zhao, Chong Luo, and Minsu Cho. Peripheral vision transformer. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, 2022. arXiv:2206.06801.

[40] Volodymyr Mnih, Nicolas Heess, Alex Graves, and Koray Kavukcuoglu. Recurrent models of visual attention. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 27, 2014. arXiv:1406.6247.

[41] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 284–292, 2005. doi: 10.1145/1062455.1062514.

[42] Avanika Narayan et al. Minions: Cost-efficient collaboration between on-device and cloud language models. In *International Conference on Machine Learning (ICML)*, 2025. arXiv:2502.15964.

[43] Nebius. SWE-agent-trajectories. https://huggingface.co/datasets/nebius/SWE-agent-trajectories, 2024. 80,036 SWE-agent trajectories; CC-BY-4.0; December 2024.

[44] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 529–540, 2007. doi: 10.1145/1315245.1315311.

[45] Laura Nguyen, Thomas Scialom, Jacopo Staiano, and Benjamin Piwowarski. Skim-attention: Learning to focus via document layout. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2413–2427, 2021. arXiv:2109.01078.

[46] NVIDIA Corporation. Variable rate shading (VRS). https://developer.nvidia.com/vrworks/graphics/variablerateshading, 2018. Turing architecture developer documentation; accessed 2026-11-01.

[47] Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M Waleed Kadous, and Ion Stoica. RouteLLM: Learning to route LLMs with preference data. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv:2406.18665.

[48] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005. doi: 10.1109/TSE.2005.49.

[49] Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. RepoGraph: Enhancing AI software engineering with repository-level code graph. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv:2410.14684.

[50] Matteo Paltenghi, Rahul Pandita, Austin Z. Henley, and Albert Ziegler. Follow-up attention: An empirical study of developer and neural model code exploration. *IEEE Transactions on Software Engineering*, 2024. arXiv:2210.05506.

[51] Anjul Patney, Marco Salvi, Joohwan Kim, Anton Kaplanyan, Chris Wyman, Nir Benty, David Luebke, and Aaron Lefohn. Towards foveated rendering for gaze-tracked virtual reality. *ACM Transactions on Graphics*, 35(6):179:1–179:12, 2016. Proc. SIGGRAPH Asia 2016.

[52] Rudrendu Kumar Paul and Sourav Nandy. AgentRouter: Heterogeneous model routing for cost-optimal multi-step agentic workflows. In *ICML 2026 Workshop on Agentic Uncertainty Quantification (AgenticUQ)*, 2026. arXiv:2609.22951.

[53] Peter Pirolli and Stuart Card. Information foraging. *Psychological Review*, 106(4):643–675, 1999. doi: 10.1037/0033-295X.106.4.643.

[54] Michael I. Posner and Yoav Cohen. Components of visual orienting. In H. Bouma and D. G. Bouwhuis, editors, *Attention and Performance X: Control of Language Processes*, pages 531–556. Erlbaum, 1984.

[55] Prithvi Rajasekaran, Ethan Dixon, Carly Ryan, and Jeremy Hadfield. Effective context engineering for AI agents. https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents, 2025. Anthropic Engineering blog, 29 Sep 2025.

[56] Yongming Rao, Wenliang Zhao, Benlin Liu, Jiwen Lu, Jie Zhou, and Cho-Jui Hsieh. DynamicViT: Efficient vision transformers with dynamic token sparsification. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 34, 2021. arXiv:2106.02034.

[57] David Raposo, Sam Ritter, Blake Richards, Timothy Lillicrap, Peter Conway Humphreys, and Adam Santoro. Mixture-of-depths: Dynamically allocating compute in transformer-based language models. *arXiv preprint arXiv:2404.02258*, 2024.

[58] Adrià Recasens, Petr Kellnhofer, Simon Stent, Wojciech Matusik, and Antonio Torralba. Learning to zoom: a saliency-based sampling layer for neural networks. In *European Conference on Computer Vision (ECCV)*, 2018. arXiv:1809.03355.

[59] Revanth Gangi Reddy, Tarun Suresh, JaeHyeok Doo, Ye Liu, Xuan-Phi Nguyen, Yingbo Zhou, Semih Yavuz, Caiming Xiong, Heng Ji, and Shafiq Joty. SweRank: Software issue localization with code ranking. In *International Conference on Learning Representations (ICLR)*, 2026. arXiv:2505.07849.

[60] Martin P. Robillard. Topology analysis of software dependencies. *ACM Transactions on Software Engineering and Methodology*, 17(4), 2008. doi: 10.1145/1348250.1348253.

[61] Paige Rodeghero, Collin McMillan, Paul W. McBurney, Nigel Bosch, and Sidney D'Mello. Improving automated source code summarization via an eye-tracking study of programmers. In *Proc. 36th International Conference on Software Engineering (ICSE)*, pages 390–401, 2014. doi: 10.1145/2568225.2568247.

[62] Ruth Rosenholtz. Capabilities and limitations of peripheral vision. *Annual Review of Vision Science*, 2: 437–457, 2016.

[63] Minjoon Seo, Sewon Min, Ali Farhadi, and Hannaneh Hajishirzi. Neural speed reading via Skim-RNN. In *International Conference on Learning Representations (ICLR)*, 2018. arXiv:1711.02085.

[64] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6):772–787, 2011. doi: 10.1109/TSE.2010.81.

[65] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. Repository-level prompt generation for large language models of code. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, 2023. arXiv:2206.12839.

[66] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling model parameters. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv:2408.03314.

[67] Seongho Son et al. SWE-Router: Routing in multi-turn agentic software engineering tasks. In *ICML 2026 Workshop on Deep Learning for Code*, 2026. arXiv:2607.00053.

[68] Christopher Theisen, Kim Herzig, Patrick Morrison, Brendan Murphy, and Laurie Williams. Approximating attack surfaces with stack traces. In *Proceedings of the 37th International Conference on Software Engineering, Software Engineering in Practice (ICSE-SEIP)*, 2015.

[69] Adam Tornhill. *Your Code as a Crime Scene: Use Forensic Techniques to Arrest Defects, Bottlenecks, and Bad Design in Your Programs*. Pragmatic Bookshelf, 2015.

[70] Karthik Vaidyanathan, Marco Salvi, Robert Toth, Tim Foley, Tomas Akenine-Möller, Jim Nilsson, Jacob Munkberg, Jon Hasselgren, Masamichi Sugihara, Petrik Clarberg, Tomasz Janczak, and Aaron Lefohn. Coarse pixel shading. In *High Performance Graphics (HPG)*, 2014. doi: 10.2312/hpg.20141089.

[71] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems 30 (NIPS 2017)*, 2017. arXiv:1706.03762; verified against the arXiv API 2026-11-01.

[72] Neil Walkinshaw and Leandro Minku. Are 20% of files responsible for 80% of defects? In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2018. doi: 10.1145/3239235.3239244.

[73] Xingyao Wang et al. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv:2407.16741.

[74] Yuhang Wang, Yuling Shi, Mo Yang, Rongrui Zhang, Shilin He, Heng Lian, Yuting Chen, Siyu Ye, Kai Cai, and Xiaodong Gu. SWE-pruner: Self-adaptive context pruning for coding agents. *arXiv preprint arXiv:2601.16746*, 2026.

[75] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

[76] Jingbo Yang et al. Ares: Adaptive reasoning effort selection for efficient LLM agents. *arXiv preprint arXiv:2603.07915*, 2026.

[77] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. arXiv:2405.15793.

[78] John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yuxiang Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2025. arXiv:2504.21798. Spotlight. Author list partly from memory.

[79] Xin Ye, Razvan Bunescu, and Chang Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014.

[80] Hongxu Yin, Arash Vahdat, Jose Alvarez, Arun Mallya, Jan Kautz, and Pavlo Molchanov. A-ViT: Adaptive tokens for efficient vision transformer. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022. arXiv:2112.07658.

[81] Peter Zatloukal. Agent-hypervisors. https://agent-hypervisor.ai/posts/agent-hypervisors/, 2025. Essay; updated 2026.

[82] Peter Zatloukal. The bitter lesson of agentic coding. https://agent-hypervisor.ai/posts/bitter-lesson-of-agentic-coding/, 2026.

[83] Peter Zatloukal. Apparently i've been doing graph engineering. https://agent-hypervisor.ai/posts/graph-engineering/, 2026.

[84] Peter Zatloukal. Thinking depth regression. https://agent-hypervisor.ai/posts/thinking-depth-regression/, 2026.

[85] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2023. arXiv:2303.12570.

[86] Yifan Zhang, Jiliang Li, Zachary Karas, Aakash Bansal, Toby Jia-Jun Li, Collin McMillan, Kevin Leach, and Yu Huang. EyeTrans: Merging human and machine attention for neural code summarization. *Proceedings of the ACM on Software Engineering*, 1(FSE):115–136, 2024. doi: 10.1145/3643732.

[87] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 14–24, 2012.

[88] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 531–540, 2008. doi: 10.1145/1368088.1368161.

[89] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005. doi: 10.1109/TSE.2005.72.

A Pilot study details

Data. Claude Code stores each session as JSON lines, with one line per message and tool-call blocks inside assistant messages; subagent transcripts are stored beside the session. We parsed 63 main sessions and their subagents across five repositories. Each tool call was mapped to zero or more file-level events: `Read` and `Grep` calls directly; `Edit`, `Write`, and the harness's own file-history records as edits; and shell commands by tokenizing the command, tracking the current simple command's head across `;`, `&&`, and pipes, stripping heredoc bodies, rejecting git pathspecs and tokens with shell syntax, and resolving each path-like argument against the repository (tracked files, files ever in history, or an unknown file with a whitelisted extension in a known directory). Reads are `cat`, `sed` without `-i`, `head`, `tail`, `grep` with a file argument, `git show`/`diff`/`log`/`blame` with a file, and similar; edits are `sed -i`, `tee`, redirections, and copy destinations; executions are interpreters and scripts. Paths under build and dependency directories are dropped. The extractor and analysis scripts (`extract_gaze.py`, `analyze_gaze.py`) are released; the per-session data stay with the author, since transcripts contain private material, and only the aggregate tables are published.

Tasks. A task is one typed user prompt and the events until the next; events are assigned by timestamp, so subagent events count toward the task that spawned them. Of 133 tasks, 67 had at least one edit, 65 of those had at least three events. Git baselines rank files by the history before the task's first prompt and are scored only on edited files that existed in git at that time (56 tasks), since a predictor cannot name a file that does not exist; the agent's-own-reads rows are scored on all edited files, which is why they are the harder comparison. The access-set analysis (Table 3) uses the 114 tasks with at least three file accesses; a mean 23% of a task's access set did not exist in git at its start (files the task created, or untracked scratch files), and those count as saccades at every budget. The co-access ranking scores a file by the number of the repository's earlier tasks in this corpus whose access set contained it (together with one of the task's first k accessed files, when conditioned), with recency as the tie-break; the first task of each repository has no prior traces and falls back to recency.

Two more machines. The laptop's and the server's sessions (Table 4) were analyzed on each machine by `collect_machine.py`, which points the scripts of this appendix at that machine's repositories and exports only numbers under labels; the raw transcripts stay where they were recorded. Their tasks are not pooled with the pilot's or with each other's, and their git-history rows are not pooled into the 19-repository figures, which already include one of the laptop's repositories and all ten of the server's. Only their per-repository concentration rows are used: their pooled rows were computed before pooled counts were keyed by repository.

Learning curve. For the measurement under E5 (`analyze_learning.py`), every pilot task with at least three accesses and an existing target is scored in chronological order per repository by the co-access ranking of Table 3, built from the tasks that finished before it started, and by recency at the same instant; tasks are binned by the number of prior traces (5, 41, 20, 30, and 18 tasks in the bins none, one to nine, ten to twenty-nine, thirty to fifty-nine, and sixty or more). The curve is dominated by R1, the only repository with more than ten traces; the per-task table is kept with the pilot data.

Learned ranker. For the measurement under E1 (`analyze_ranker.py`), every file that existed in git at a task's start is a candidate, labeled by membership in the task's access set and, separately, its edit set; the features are listed in the script. A logistic regression with class weighting and an L2 penalty is fit once per label on the pooled training tasks (the first 60% of each repository's tasks in time; 1,198 access-set positives among 25,057 candidate rows) and scores the later tasks; prior-trace features see only tasks that finished before the task started. The pilot's prompts are stored truncated to 400 characters; with the full prompt text in their place the lexical feature gains weight and the ranker loses a point (Recall@10 of the access set 0.51), so the truncation is not what holds it back. The test set is dominated by R1 (31 of 46 tasks).

Ceiling accounting. For the ceiling of Section 3.2 (`analyze_ceiling.py`), every Read call, Grep on a file, and shell read in the transcripts is paired with its tool result, whose length in bytes is what the model saw (a shell command that read several files splits its output among them). A read is charged once per later assistant turn in its transcript, since the harness re-sends every tool result on every later call, and converted to tokens at four bytes a token; the task's processed context is the sum over its assistant calls of input, cache-read, and cache-creation tokens, subagents included. Files are classified at the task's first prompt: edited in the task, new in the task, a map file, or inside or outside the recency top-k; for a peripheral file outside the edit set, the stub size comes from the pilot's signature extractor applied to the file's content at the last commit before the task, and a file the extractor does not handle counts as a one-line name. The per-task table is kept with the pilot data; the aggregates are in the released results file.

Additional measurements. Sessions spawned 146 subagents; 111 were review subagents launched at the highest effort setting, 28 inherited the session's model and effort, and 1 used a cheaper model. Over all sessions the agent made 10,290 shell calls (5,001 touching no repository file), 1,080 `Read` calls, 557 `Edit` calls, and 354 `Write` calls; 6,072 of 6,614 read events came from shell commands. Across sessions, the median Jaccard similarity between consecutive sessions' touched sets was 0.13 to 0.19 in four repositories and 0.49 in the ML pipeline, where one feature was being built over several days.

B Live replay pilot

To exercise the intervention apparatus we replayed three historical commits from R1 as tasks: for each, the parent commit was checked out into a throwaway worktree, a task description written from the commit's intent (not its diff) was given to a headless agent, and the agent's gaze, edit set, cost, and turns were recorded, under four arms: the plain task; the task with a ranked list of the five most recently committed files at the parent (a cheap predictor); and each of those with a cheaper model. Table 9 reports the runs. With three tasks and one run per cell this is a smoke test of the runner, not evidence for or against H3; it establishes that the apparatus measures what Section 5 needs and what a run costs. Three observations, each from a single run and offered as such. Every arm edited every file the historical commit had edited (recall 1.0 in all 12 runs), and the frontier arms on the second task also factored the validation into a script with its own test, which the historical commit had not done. The recency hint, which named only half or a third of the gold files on two of the tasks, cut the frontier model's turns on the two larger tasks (44 to 35 and 56 to 38) and their cost by 16% and 23%, and changed nothing on the smallest. The cheaper model finished in 10 to 20 turns at $0.18 to $0.29 against $0.67 to $1.76 for the frontier model, and reached the same files; whether its patches are correct was not tested here, since the replay has no hidden tests, which is the gap E2 and E3 close. Total model spend for the twelve runs was $9.04; wall time ran from 22 seconds to under six minutes per run.

Table 9: Live replay pilot on R1 (measured; one run per cell). Recall and precision are of the agent's changed files against the historical commit's files. Orientation reads are distinct files read before the first edit.

Task	Arm	Model	Turns	Distinct reads	Orient. reads	Recall / precision	Cost ($)
T1	plain	opus	22	4	3	1.00 / 1.00	0.67
T1	recency hint	opus	22	4	4	1.00 / 1.00	0.68
T1	plain	sonnet	10	2	2	1.00 / 1.00	0.18
T1	recency hint	sonnet	13	3	3	1.00 / 1.00	0.19
T2	plain	opus	44	8	7	1.00 / 0.40	1.71
T2	recency hint	opus	35	7	6	1.00 / 0.40	1.44
T2	plain	sonnet	19	4	3	1.00 / 0.40	0.28
T2	recency hint	sonnet	18	3	2	1.00 / 0.40	0.26
T3	plain	opus	56	12	11	1.00 / 1.00	1.76
T3	recency hint	opus	38	11	10	1.00 / 1.00	1.35
T3	plain	sonnet	20	6	6	1.00 / 1.00	0.29
T3	recency hint	sonnet	13	7	7	1.00 / 1.00	0.23

Foveated worktree pilot. With the same three tasks we exercised the mechanism of Section 4.2 end to end. The bytes on disk stayed a full checkout; a regular-expression extractor wrote a signature view of every file outside the fovea (declarations for code, headings and frontmatter for prose, top-level keys for configuration), a pre-tool hook passed to the headless harness redirected the model's reads of those files to their views and refused to edit them, and a `fovea materialize` command printed a file in full, unlocked it, and logged the saccade. Shell readers were removed from the allowed tools so that reads went through the hook (not airtight: `node -e`, `git show`, and `find -exec` remained; a hand review of the list `scan_shell_reads.py` draws from every shell call in the 85 foveated runs of this appendix, each segment that names a tracked file outside the run's fovea and contains a reader or a write, found no command that showed the model the content of a stubbed file and 2 runs in which a node one-liner read stubbed files inside a subprocess, once to write a modified copy of a demoted file for a test and once to run a check over the posts, showing the model only the result; the review gate's independent scan found one more, a one-liner that appended a test line to a stubbed post and restored it, an edit the hook, which sees only the edit tools, could not refuse); search still saw full files. Each task had a hidden test, run after the agent finished: the gold commit's own stats test for T1, three malformed-date builds that must fail and a restored build that must pass for T2, and a passing build with the hard-coded post lists gone for T3. Four arms, three frontier-model runs each (the first run of every cell in the morning, two more the same evening, when the fourth arm was added): the full worktree with shell readers disabled (the control for that restriction); an oracle fovea of the gold files, their direct import neighbours, and the map files (5 to 9 files at full text, 50 to 62 stubs); the oracle access set, what the first full-worktree run of the same task read plus the gold

and map files (8 to 13 files); and a predicted fovea of the ten most recently committed files plus the map files (12 to 14 at full text), which held every gold file for T1 and T2 and two of three for T3. Table 12 reports the cells. All 36 runs passed their hidden test and every arm edited every gold file. Over three seeds the oracle fovea cost 16% less than the full worktree ($1.00 against $1.19 per task; seed means per task 9%, 25%, and 10%), the oracle access set 11% less, and the predicted fovea 6% less; single-seed pairings of the same cells range from -10% to +40%, which is why one run per cell had put the oracle at 18% and the access set at 6% an hour apart, and why no saving in this appendix is quoted without its seeds. Costs are the agent's spend end to end, with the stub reads and materialization round trips inside them; view generation was a script and the ranking was `git log`, so the orchestration cost nothing and the saving is net of the overhead the mechanism adds. Across the 27 foveated runs the agent read a signature view 53 times and materialized 27 files: the view sufficed for 26 of 53 peripheral reads, a first point on the acuity curve of Section 5. The saccades were of two kinds. Under the predicted fovea on T3, the agent materialized the gold file the predictor had missed and edited it in every seed, the graceful-degradation path of H7 at the cost of one tool call each time. Most of the rest were templates, data, and scripts that the verification scripts assert against or that the task's checks run (the index and sitemap templates, the posts list, the deploy script, the check scripts), which no import graph connects to the edited files: an oracle built from gold files and imports is not the oracle access set, and the oracle access-set arm, which holds those files at full text, saccaded a third as often and saved less, because a file at full text is paid for on every turn whether or not it is read again. The last kind was the repository's review and security logs, which the agent opened to check house conventions: maps for the agent that no import graph connects to the task and that the pilot's recency ranking excludes by construction (at T2's parent the two review logs were the third and fourth most recently committed files, and the hint drops them by construction), the pilot's instance of the map-file problem.

Oracle access set and demotions (E10-lite). The oracle access-set arm (the files the first full-worktree run of the same task had read, plus the gold files and the map files, at full text, everything else a signature stub; 8 to 13 files) was added for the demotion runs and then seeded with the rest; Table 12 has it 11% under the full worktree over three seeds, between the import-neighbour oracle and the predicted fovea, a strict superset of the import oracle that saved less. Then, for each of the 15 non-map files of that fovea, one run with the file demoted to its signature view and materialization allowed, the labels of E10 on real tasks. No demotion flipped a hidden test. Nine were materialized, four were read as a view and never materialized, two were never read; every demoted file the task edits was materialized before the edit, since the hook refuses to edit a view; and the cost of a demotion against the undemoted arm ran from -17% to +43% (median -6%), noise except on the 15-turn task, where one extra round trip is a third of the run. On these tasks the signature tier was safe as a periphery, the saccade recovered every file the agent decided it needed, and the agent decided it needed full text for most of what it opened, including files it did not edit.

Orientation ablations (E6-lite). The same three tasks, three runs each, under two more arms (`e6-results.md`): the worktree with the instruction files (`CLAUDE.md`, `README`) deleted before the run, so the harness injects no map; and the full worktree with a per-task foveal map prepended to the prompt, the recency top-10 files at the parent each shown as its signature view. Against the plain worktree's nine runs (orientation 11.2 accesses over 7.2 distinct files, 0.63 million tokens before the first edit, $1.19), the map arm oriented in 12.1 accesses over 6.9 files, 2% fewer tokens before the first edit, and 6% less cost; the no-map arm in 11.9 accesses over 7.9 files, 17% fewer tokens before the first edit, and the same cost. All 27 runs passed their hidden tests. The map did not shorten orientation on these tasks, and the harness's own map was not what kept it short either; the orientation these tasks need is a dozen accesses, and a hundred-file repository does not give a map much to shorten.

Adversarial tasks (E8-lite). Ten further tasks were drawn from the same repository's history by a script that keeps commits in which git recency ranks at least one changed file outside the top twenty at the parent and the commit's subject line does not name it (the full message names the file for four of the ten), with prompts written from the messages without naming files and a hidden check per task written before the runs (Table 10). Three arms: the full worktree and the predicted fovea (recency top-10 plus the map files, the pilot's own predictor) with three frontier-model runs each, and the predicted fovea plus an exploration slice (the three most-imported files and a seeded random tenth of the tree) with one. Under the predicted fovea, seven tasks had 8 gold files outside it; the other three were rescued by the map rule, because their obscure file is `.eleventy.js`, which the design

always shows in full, and they serve as controls. Every run edited every gold file. All 24 misranked files (eight per seed) were materialized before the agent's first edit; the predicted fovea spent 57 saccades over its thirty runs and cost 5% less than the full worktree per task, each task against its seed mean (median 1%); the exploration slice caught one of the eight by itself and cost 6% more (median 2%). All 70 runs passed their hidden checks; three of the ten checks had rejected valid work (one looked for brackets as literal text where the agents had used spans, one for a particular class name where an agent had used another for the same rule, one did not accept the linkifier's `http://` for a bare domain) and were corrected after the runs, and the affected runs were re-scored from their saved patches without rerunning the agent. The hostile regime, on this repository, costs saccades and nothing else.

Table 10: Adversarial tasks on R1 (measured; three frontier-model runs per cell, means, cost with its range; every run edited every gold file and passed its hidden check). *Sacc.*: saccades per run under the predicted fovea; the last column is the predicted fovea's mean cost against the full worktree's, and its mean row is the column's mean (the ratio of the two arms' mean costs is -10%). The exploration arm (one run per task) is in the results file.

		Full worktree		Predicted fovea			
Task	Gold	Turns	Cost ($)	Sacc.	Turns	Cost ($)	vs. full
adv1	1	10.3	0.31 (0.28 to 0.35)	1.0	9.0	0.29 (0.28 to 0.31)	-6%
adv2	1	8.3	0.29 (0.28 to 0.31)	1.0	8.3	0.29 (0.29 to 0.30)	+2%
adv3	3	21.3	0.41 (0.39 to 0.44)	2.0	22.0	0.43 (0.41 to 0.45)	+4%
adv4	2	33.0	0.76 (0.72 to 0.81)	2.7	38.0	0.87 (0.80 to 1.00)	+15%
adv5	2	17.0	0.51 (0.48 to 0.53)	1.3	16.7	0.51 (0.50 to 0.54)	+1%
adv6	3	48.0	1.92 (1.59 to 2.17)	3.3	36.0	1.44 (1.29 to 1.59)	-25%
adv7	2	38.7	0.92 (0.76 to 1.05)	4.7	34.7	1.00 (0.72 to 1.16)	+8%
adv8	3	47.0	1.28 (1.09 to 1.49)	1.7	48.0	1.24 (1.04 to 1.53)	-3%
adv9	2	36.3	0.93 (0.76 to 1.15)	1.3	28.7	0.76 (0.71 to 0.80)	-19%
adv10	2	37.0	1.41 (1.13 to 1.70)	0.0	23.7	1.02 (0.93 to 1.13)	-27%
mean		29.7	0.87	1.9	26.5	0.79	-5%

Table 11: Demotion pilot on R1 (measured; one frontier-model run per cell). Each row demotes one file of the oracle access-set fovea to its signature view; gold marks files the historical commit edited; view reads are reads answered by the signature view; cost is against the undemoted oracle access-set arm of the same task.

Task	Demoted file	Gold	View reads	Materialized	Saccades	Turns	Hidden test	Cost ($)	vs. arm
T1	`lib/stats.mjs`	yes	1	yes	1	15	pass	0.71	$+33\%$
T1	`scripts/check-stats.mjs`	yes	1	yes	1	15	pass	0.76	$+43\%$
T2	`CODEREVIEW.md`	no	0	no	0	33	pass	1.22	-12%
T2	`deploy.sh`	no	0	no	0	39	pass	1.27	-8%
T2	`scripts/check-seo.js`	no	1	yes	1	47	pass	1.30	-6%
T2	`scripts/verify-live.js`	no	1	yes	1	42	pass	1.26	-9%
T2	`src/posts/posts.json`	no	1	no	0	38	pass	1.14	-17%
T3	`scripts/check-seo.js`	yes	1	yes	2	50	pass	1.39	-1%
T3	`scripts/verify-live.js`	yes	1	yes	2	40	pass	1.19	-15%
T3	`src/all.njk`	no	1	no	1	53	pass	1.63	$+17\%$
T3	`src/index.njk`	no	1	yes	2	46	pass	1.37	-2%
T3	`src/posts/first-fable.md`	no	1	no	1	41	pass	1.25	-10%
T3	`src/posts/posts.json`	no	1	yes	2	45	pass	1.25	-10%
T3	`src/robots.txt`	no	1	no	1	47	pass	1.32	-5%
T3	`src/sitemap.njk`	no	1	yes	3	59	pass	1.67	$+20\%$

C History-only predictors on SWE-bench Verified

Before any model run, we scored three checkout-free rankings of the gold patch's files on all 500 Verified instances: git recency and git commit frequency over the base commit's ancestry, and a lexical path match that scores a file by the issue text's identifiers and dotted module paths against its path components, with unnamed files following in recency order. Candidates are the files in the base tree (median 4,416 per instance); 71 instances have multi-file patches (mean 1.24 gold files). Table 13 gives the result. The regime claim of Section 3.2 holds with room to spare: recency, which recovered two thirds of the next edit set on an active single-developer repository, recovers a tenth of it here, and the pre-registered target of $0.21^{\dagger}$ was itself too generous. The path match is the surprise: half of the gold files are named, in effect, by the issue, which is the memorization-free floor any learned

Table 12: Foveated worktree pilot on R1 (measured; three frontier-model runs per cell). Fovea and stubs are file counts; view reads are reads answered by a signature view; saccades are materializations; turns, view reads, and saccades are means over the three runs; cost is the mean with the range across runs; the hidden test is the task's own.

Task	Arm	Fovea	Stubs	Gold in fovea	Turns	View reads	Saccades	Hidden tests	Cost ($)
T1	full worktree	97	0	2/2	17.0	0.0	0.0	3/3	0.67 (0.60 to 0.71)
T1	oracle fovea	9	62	2/2	16.0	0.0	0.0	3/3	0.61 (0.58 to 0.66)
T1	oracle access set	8	63	2/2	15.0	0.0	0.0	3/3	0.59 (0.50 to 0.66)
T1	predicted fovea	14	58	2/2	14.3	0.0	0.0	3/3	0.66 (0.53 to 0.73)
T2	full worktree	82	0	2/2	38.0	0.0	0.0	3/3	1.47 (1.35 to 1.65)
T2	oracle fovea	5	52	2/2	35.3	4.0	1.0	3/3	1.10 (0.99 to 1.26)
T2	oracle access set	10	47	2/2	34.3	0.3	0.0	3/3	1.23 (1.03 to 1.50)
T2	predicted fovea	12	45	2/2	40.0	1.3	0.3	3/3	1.35 (1.27 to 1.48)
T3	full worktree	82	0	3/3	47.7	0.0	0.0	3/3	1.44 (1.31 to 1.56)
T3	oracle fovea	7	50	3/3	42.3	5.0	3.0	3/3	1.29 (1.22 to 1.40)
T3	oracle access set	13	44	3/3	45.7	2.0	1.3	3/3	1.36 (1.25 to 1.49)
T3	predicted fovea	14	44	2/3	44.3	5.0	3.3	3/3	1.37 (1.25 to 1.48)

localizer must clear and part of what a code ranker's 0.86† is made of. The scripts and per-instance scores are released with the extractor.

Table 13: History-only and path-match predictors of the gold edit set on SWE-bench Verified (measured, 500 instances). Below: Recall@10 by repository.

Ranking	Recall@5	Recall@10	Recall@20	AUFC
Git recency	0.07	0.11	0.15	0.89
Git frequency	0.11	0.19	0.27	0.91
Lexical path match	0.40	0.51	0.59	0.95

Repository	n	Rec.	Freq.	Path
django	231	0.09	0.19	0.47
sympy	75	0.07	0.12	0.56
sphinx	44	0.14	0.11	0.30
matplotlib	34	0.18	0.30	0.60
scikit-learn	32	0.11	0.03	0.73
astropy, xarray	44	0.08	0.32	0.55
pytest, pylint	29	0.12	0.05	0.50
requests, seaborn, flask	11	0.55	0.77	0.59

The same rankings on the developer's repositories. For the breadth claim of Section 3.2 (`analyze_history_repos.py`), every non-merge commit of each repository is a task in chronological order after the repository's first ten commits, bulk commits over twenty files are skipped, the gold set is the commit's files that existed before it, and recency, frequency, and the previous commit's files are ranked from the commits before it; the label-to-path map stays outside the released code, since 13 of the 19 repositories are proprietary and appear only as labels. Over the 15 repositories with at least twenty scored commits, the commit-weighted Recall@10 is 0.55 for recency, 0.59 for frequency, and 0.55 for the previous commit's files, which head the recency ranking by construction, so those two coincide except where commit times tie or run backwards; the repository means are the same to two decimals for the first two; Recall@10 by commit position is flat (0.53 to 0.57 from the tenth commit to the five-hundredth and beyond), so history helps from a repository's first weeks and does not improve with its age. A commit is a coarser task than a prompt (one prompt can end in several commits and several prompts in one), which is why these numbers sit beside the transcript pilot's rather than in its table.

Text-only arm. The memorization control of E1 (`memorization_control.py`, after Liang et al.) gives a model the issue text and the repository's name, no checkout and no tools (the headless harness with an empty tool allowlist and a single turn; a rerun of four instances with the event stream recorded no tool call), and asks for up to ten repository paths the fix will edit, most likely first; the score is recall of the gold edit set at exact path match, and at basename match as a relaxed check (0.90 either way). One call per instance with a cheap model, $2.67 for all 500. The calls were made from inside this paper's repository, so the harness injected its own instruction file, which says nothing about the

benchmark repositories; the allowlist was emptied rather than the tools removed, which the harness honours within one turn, and the recorded transcripts contain no tool call (later runs disable the tools outright and run from an empty directory). The protocol is not Liang et al.'s (ten guesses here against their own task definition), so only the direction is comparable.

D Pre-registration

The hypotheses, predictions, and falsification criteria below were committed to the project repository on 2026-11-01 before any experiment beyond the pilot was run; the pilot's own H1 prediction was written while a first, unfiltered run was already visible and is marked as such in the plan. H8 and H9 were added the same day, after an external review of the first draft and before any run that tests them.

- H1 **Concentration.** Pooled Gini ≥ 0.6; top decile $\geq 50\%$ of reads; per-session coverage $< 10\%$ in repositories of ≥ 200 files. Falsified by Gini < 0.3 or top decile $< 25\%$. *Pilot: supported.*
- H2 **Predictability.** Learned ranker beats BM25 by ≥ 10 points Recall@10 on SWE-bench Verified; orientation conditioning ($k = 5$) adds ≥ 15 points. Falsified if the best pre-run predictor is within 5 points of BM25 and orientation adds < 5. *History baselines measured on all of Verified (2026-11-01): recency 0.11, frequency 0.19, lexical path match 0.51 at ten files; the recency target of 0.21 was too high. Pilot: a co-access ranking from prior traces adds 5 points over recency on the remaining access set after five accesses, below the 15 the hypothesis asks of full conditioning.*
- H3 **Foveation benefit.** Context foveation within 2 points of uniform at $\geq 30\%$ fewer input tokens; compute foveation at $\geq 90\%$ of the all-frontier pass rate at $\leq 50\%$ cost. Falsified if no foveated arm reaches 95% of the uniform-strong pass rate at $\leq 70\%$ cost on both benchmarks.
- H4 **Risk weighting.** $\geq 20\%$ fewer escaped defects at equal cost. Falsified by $< 5\%$.
- H5 **Learning.** Recall@10 rises ≥ 15 points from 0 to 100 traces; saccades fall $\geq 40\%$. Falsified if within 5 points.
- H6 **Orientation.** A foveal map cuts orientation accesses $\geq 40\%$ with pass@1 unchanged within 2 points. Falsified by $< 10\%$. *E6-lite (three pilot tasks, three seeds, Appendix B): falsified in the pilot regime; the map changed orientation accesses by $+8\%$. The claim stands only for repositories where orientation is long.*
- H7 **Graceful degradation.** A wrong fovea costs < 5 points and $\leq 25\%$ cost. Falsified by ≥ 10 points. *A scheme that fails this is not deployable regardless of H3.*
- H8 **Adversarial attention.** On tasks built so that the cheap signals mis-rank the needed file, FTW stays within 5 points of the uniform arm and materializes that file before the first edit in $\geq 70\%$ of runs. Falsified by a drop ≥ 10 points. *E8-lite (ten tasks, three seeds in the full-worktree and predicted-fovea arms, one in the exploration arm, Appendix B): supported; all 24 misranked files materialized before the first edit, hidden checks 30 of 30 in each seeded arm and 10 of 10 in the exploration arm.*
- H9 **Learning under intervention.** A predictor trained on foveated traces with the calibration stream matches one trained on full-worktree traces within 1 point of pass@1 under budget; without the stream it is ≥ 2 points worse while scoring higher on its own access-set recall. Falsified if the stream recovers < 1 point of that gap, or if the gap does not appear.

E Human-reviewed subset

For $60^{\dagger}$ tasks ($30^{\dagger}$ FTW, $30^{\dagger}$ uniform, blinded), a reviewer read the agent's patch and the files it had at reduced resolution and judged whether a peripheral summary had been relied on incorrectly. Target: $2^{\dagger}$ such cases in the FTW arm, both caught by hidden tests; $0^{\dagger}$ in the uniform arm. The instrument saturates at about thirty items a sitting [83], which bounds its size.

F Prompts and hints

The orientation hint used in the live pilot and in E6 reads: "A predictor ranked these files as the most likely to be involved (most likely first): ⟨list⟩. Use the list to orient; verify before relying on it." The foveated worktree needs no prompt: the agent sees an ordinary checkout, and a file below full text carries a one-line header stating its tier and that reading it in full is one tool call away.