Agentic Execution Optimization

Arun Kejariwal

“Are you AI-Native?" is now asked constantly, but the early signals were low-SNR - "TokenMaxxing" became a fad and subsided once costs ballooned. The durable signal is productivity: diffs/developer, agent-driven reviews and approvals, lower revert rates, faster SEV mitigation [1, 2], alongside clear traction in agent-driven autoresearch [3, 4, 5, 6, 7, 8].

As agentic execution has gained traction - driven by a steady stream of success stories - the unit of work has evolved from simple prompts to complex, high-level tasks. Examples include chip design [9], GitHub Spec Kit, synthetic data generation [10, 11, 12], proving/disproving conjectures[1] etc. In such scenarios, a frontier model alone is insufficient; a robust harness is equally important - arguably more so. Consequently, there has been a lot of chatter around the harness [17, 18, 19, 20, 21]. In fact, harness optimization[2] - which includes, but is not limited to, planning, memory and tool use management, reasoning - has gotten attention as well, e.g., MetaHarness, Self-Harness**.** In general, agent execution optimization is multi-faceted:

- *Output quality and consistency*
    - The latter stems from the fact that the output can vary from one run to another.
- *E2E latency*
- *Total Cost of Ownership (TCO)*
    - A heterogeneous - different vendors and different generations for a given vendor - setup across compute (CPU, GPU, TPU, NPU), memory bandwidth and capacity, network bandwidth, disk capacity

From an operational standpoint, the following metrics are of interest:

- *Planning efficiency*
- *Tool efficiency* (whether tools were used appropriately)
- *Agent recovery rate* (in the wake of, say, tool failure, network failure)
- *Agent redundancy* (owing to potential duplicative work across agents)
- *Coordination overhead in a multi-agent scenario*

Interestingly, there are a lot of parallels between problems that have been addressed in the realm of compiler-driven program optimization - e.g., auto-parallelization, auto-vectorization and profile-guided optimization - and optimization of agentic execution. A subset of them are discussed below.

Task Definition

To exploit coarse-grain program parallelism, compilers break down an input program into a task graph. Typically, tasks are defined at the loop level and/or function level. Task granularity directly impacts performance. For instance, small tasks make exploiting thread-level parallelism unprofitable owing to the threading overhead. Likewise, in a message passing context, communication overhead may trump the performance gain harnessed from parallel execution. Additionally, task granularity may affect HW mapping - should a given task be executed on a CPU or GPU or NPU for best performance - and global scheduling, which in turn impacts system utilization.

The above has a direct parallel in agentic execution context. Let’s consider a typical pipeline in the context of autonomous driving: (i) Data Ingestion & Sync - Sensor Aggregation and Temporal & Spatial Alignment (ii) Quality Filtering & Pre-processing - Noise Suppression and Dynamic Calibration (iii) Perception - multi-modal sensor fusion, 3D Bounding Box Detection and Semantic Segmentation & Intent Classification (iv) Motion

[1] Recently, leading with a higher level description of the planar unit distance problem (see page 3 of [13]), an unexpected connection between algebraic number theory and discrete geometry was discovered and helped disprove the conjecture [14]. Likewise, the Jacobian conjecture was disproved using Fable 5 [15] and Dinitz-Garg-Goemans conjecture was reportedly disproved using GPT-5.6 Pro [16].

[2] In [22], the authors share that providing repository-level context files does not generally improve task success rates, while increasing inference cost by over 20% on average. This observation holds across different LLMs, coding agents, and for both LLM-generated and developer committed context files. Interestingly, repository overviews, although popular and recommended by model providers, are not helpful.

Tracking & State Estimation - Deep Multi-Object Tracking (MOT) and Kinematic Estimation (v) Prediction - Trajectory Distribution Generation, Interaction Modeling and (vi) Decision Making, Planning & Control. Each stage is mapped to an agent and the underlying tasks are mapped to sub-agents. The same granularity trade-offs apply to agent execution. In particular, in scenarios where agents spawn sub-agents, it's important to bear in mind that sub-agents do not inherit the context and inheritance is replaced by retrieval or persistent artifacts (refer to Appendices A1 and A2 for further discussion).[3] Re-learning overhead in a hierarchical multi-agent setup can be reduced in the following fashion:

- *Front-load stable context into `CLAUDE.md`, not the prompt.* Anything every sub-task needs — architecture, conventions, key file locations, build/test commands — goes here. It's re-read at session start and re-injected after every `/compact`, so it survives compaction and is automatically inherited by the main agent (subagents also load project memory).
- *Spawn only along clean seams*. A boundary that requires you to paste half the parent's reasoning into the prompt is the wrong boundary — do it inline instead.
- *Return summaries, not raw material.* Each subagent should hand back a compressed result. That's the whole economic point — 40 files read, three lines back — and it keeps the parent's context from filling up, which delays compaction.
- *Lean on compaction and auto-memory for the long tail.* As the main context fills, `/compact` summarizes history while `CLAUDE.md` and auto-memory retain the durable facts, so the essentials aren't lost mid-task.

Cost Model

As with thread-level parallelism, vectorization of every loop, assuming feasibility, does not guarantee speedup (refer to Appendix A3 for a brief discussion of the overheads). Compilers therefore gate on explicit cost models - GCC's vectorizer and auto-parallelizer apply profitability thresholds (e.g., minimum iteration counts) before emitting parallel/vectorized code.

Claude Code, by contrast, has no formal cost model: granularity is decided heuristically from signals like whether the sub-task is self-contained, whether its output is summarizable, and whether it fans out across many files/paths. Read-only exploration and parallelizable work are the natural candidates. Mapping agents to heterogeneous hardware will require cost models that account for the compute, memory, and bandwidth constraints of the target silicon.

Fusion

Loop fusion amortizes parallelization/vectorization overhead. The agentic analog: spawning a sub-agent carries fixed overhead - re-sending the system prompt and context, prefill, network round-trip - that doesn't shrink with the work done, so a trivial-work agent has a terrible useful-work-to-overhead ratio. Fusion in agentic execution has the following flavors:

- *Step/prompt fusion:* collapse several tiny sequential sub-tasks into one prompt ("do X, then Y, then Z, return all three") so one call's fixed overhead covers all of them.
- *Role merging:* MALBO's Bayesian search *discovers* hybrid agents that combine what were separate specialist roles, because two thin agents cost two full context-loads; one merged agent costs one.
- *Batching:* issue independent tool calls or queries in a single request rather than N round-trips.

There's a key distinction: in the agentic context, dependency analysis to ensure that fusion is safe is not needed. Fusion is carried out when the orchestrator (or an optimizer such as DSPy/MALBO) judges the overhead ratio is bad, and the risk is that over-fusing one giant prompt hurts quality (the model attends worse to a bloated instruction).

---

[3] In general, as much as a 1M-token context is useful as it reduces retrieval frequency and allows larger chunks of work to stay in working memory, for truly large tasks, the dominant mechanism is hierarchical decomposition and memory management, not brute-force context stuffing.

Profile-Guided Optimization (PGO)

For cases where a loop's trip count is not known at compile time, a compiler may multiversion the loop in the following fashion, with branch probabilities coming from PGO:

```
if (trip_count < 1000)
    scalar_loop();
else if (trip_count < 100000)
    simd_loop();
else
    openmp_loop();
```

The optimization process becomes: Compile → Profile[4] → Recompile → Deploy. The direct analog of "keep several versions of the loop and dispatch the best" in agentic execution is keeping several *ways to answer* and dispatching per query. Specifically:

- *Cascades* run cheap→expensive models in order with early exit - this is almost exactly a PGO cascade. FrugalGPT is the canonical instance: a weak model answers, a learned *quality estimator* scores it, and a stop judge decides whether to escalate. That stop judge *is* the runtime dispatch predicate.
- *Routers* pick a variant up front instead of sequentially. RouteLLM learns the strong-vs-weak choice from preference data (refer to Appendix A4 for further discussion); AutoMix adds self-verification; Route-to-Reason (RTR) and Router-R1 route not just the *model* but the *reasoning strategy* (direct answer vs. chain-of-thought vs. tool loop) - i.e., they choose "scalar vs vectorized" over the reasoning method, and report ~60% token reductions.

PGO's binary instrumentation has a direct analog: LLM observability/tracing (OpenTelemetry-for-LLMs, AgentOps-style) capturing dependency structure, per-node token counts, latency, tool-call outcomes, context sizes, and success/failure. These traces feed back into *rewriting the workflow*: for example, current schedulers often use FIFO or topological order, whereas a PGO-style approach would learn that search and coding can overlap safely, thus transforming:

```
A → B → C  to  A
               ├── B
               └── C
```

thereby, reducing wall-clock time. This is directly analogous to instruction scheduling or out-of-order execution in the context of program execution on CPUs. The tunable surface of Agent PGO is much wider than its compiler counterpart: agent count, communication topology, role assignment, model selection per node, and the prompts themselves. The following makes agent "PGO" harder than the compiler version:

- Every variant is *correct* in PGO; here they differ in *quality*. Serial and SIMD loops produce identical results, so profiling optimizes pure cost. Weak-model vs strong-model produce *different answers*, so the profile signal is multi-objective (cost × latency × quality) and the quality axis is hard to measure online - you need verifiers, judges, or self-consistency as proxies, and those cost tokens themselves. The dispatch predicate is statistical, not a guaranteed-safe transform.
- Non-determinism. The same node with the same input can behave differently run to run, so profiles are distributions, not counts. Optimization has to be robust to variance rather than keyed to a fixed hot path (refer to Appendix A5 for further discussion).

Speculative Execution

To boost performance, compilers support speculative execution in multiple flavors: *control/branch speculation* hoists a load above a branch, *data speculation* issues a load ahead of a possibly-aliasing store, and *thread-level speculation* parallelizes loops whose dependencies can't be proven absent - each paired with a

[4] In general, a profile captures: hot loops, branch frequencies, cache misses, memory bandwidth, vectorization effectiveness.

check-and-recover path that repairs state on a misspeculation. The economics are uniform: speculation pays when the predictor is accurate and the rollback cost is bounded, trading wasted work on mispredictions for hidden latency. (Unlike the multiversioning in PGO, nothing here selects a "best" variant - the point is to *start early*, not to choose.)

The tightest agentic analog lives at the token level. Under speculative decoding [23, 24], a small draft model proposes *k* tokens, which the target model verifies in a single parallel forward pass; accepted tokens commit and the first rejected token triggers a rollback to that position. The structure is identical to branch speculation - predict, execute ahead, squash on mismatch - and it hides the target model's per-token latency. Notably, this speculation is *lossless*: the accepted stream matches the target model's own distribution exactly.

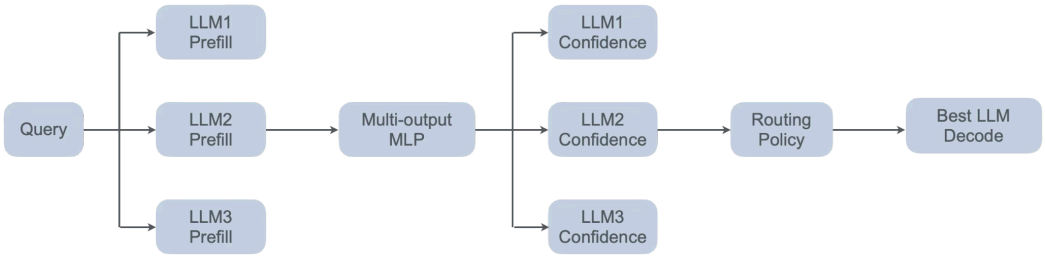
One level up, agents speculate over the task DAG. Rather than wait for the current step to resolve, the orchestrator can predict the likely next task and start that worker early - pre-warming a retrieval, or kicking off the probable tool call - and discard the result if the branch isn't taken:

CPU: Predict branch → Execute ahead → Rollback if wrong

Agent: Predict likely next task → Start worker early → Discard if unnecessary

The trade is the same CPUs make - wasted tokens on a misspeculation against latency saved on a correct one - but with a softer rollback, since a discarded sub-agent leaves no shared state to repair.

The efficacy of speculative execution rests on the quality of the underlying predictor. In agentic execution, prefill-activation routing is one such predictor. Classic routers rely on semantic or intent-based signals - query features and domain descriptions - and ignore the routed models' own generative limitations. Prefill-activation routing uses the model's activations as the routing signal instead: by the time the model has finished reading the prompt - before it generates a single token of the answer - its internal state already reflects whether it is likely to answer correctly. A lightweight classifier (e.g., a multi-output MLP) reads that state, estimates each candidate model's chance of success, and routes the query to the cheapest model likely to get it right (see below).



This is the near-exact analog of value prediction: estimate the outcome cheaply before committing to the expensive path.

HW Scheduling

In compilers, mapping tasks to hardware - CPU vs. GPU, across types and generations - directly affects per-task and overall latency. (A locally sub-optimal mapping can still lower overall latency by reducing contention.)

Likewise, in the context of agentic execution, prefill-decode disaggregation is commonly employed as prefill is compute-bound and decode is memory-bandwidth/capacity-bound. DistServe and Splitwise split the phases across separate GPU pools for 2–7× throughput, with Splitwise explicitly pairing H100 and A100 for energy efficiency. High-FLOP parts (B200/H200) act as the prefill intake engine; decode goes to large-memory, high-bandwidth, cheaper parts. Mooncake adds KV-cache-centric SLO-aware scheduling; Cronus does partially-disaggregated prefill for heterogeneous clusters; Tessera pushes disaggregation down to *kernel granularity*.

Summary

The following table summarizes the parallels across different axes.

Axis	Compiler optimization	Agentic-execution equivalent
Optimization objective	Minimize runtime / code size (single, measurable, deterministic)	Jointly minimize cost (tokens/$) × latency × quality - multi-objective, quality hard to measure
Unit being optimized	Loop, basic block, function, call graph	Node/step in an agent DAG: an LLM call, tool call, or subagent
Instrumentation / profile run	Instrumented build emits edge/block counts; or sampling profiler	LLM/agent tracing & observability: per-node tokens, latency, tool outcomes, success/failure
What the profile records	Execution counts, hot paths, branch bias, value profiles	Token counts, wall-clock, tool-call success, escalation events, per-node solve rate
Multiversioning (variants kept)	Serial / auto-parallel (OpenMP) / SIMD versions of a loop	Weak vs. strong model; direct-answer vs. CoT vs. tool-loop strategy; single-agent vs. multi-agent team
Dispatch mechanism	Runtime CPU-feature check / trip-count guard picks a version	Router (pick variant up front) or cascade (cheap→expensive with early exit)
Dispatch predicate / signal	Loop trip count, alignment, target ISA	Learned quality estimator / stop judge; router trained on preference data (which answer wins)
Predict-before-execute	Branch prediction, value profiling	Prefill-activation routing: probe residual-stream hidden states at the last prompt token to predict answer correctness before decoding
Offline "recompile with profile"	Rebuild with counts → better inlining, layout, spill decisions	Trace-driven workflow compilation: rewrite prompts, topology, roles, per-node model
Granularity adjustment	Loop fusion (merge tiny bodies to amortize parallel overhead); inlining; loop splitting	Step/prompt fusion (collapse thin steps into one call), role merging, batching tool calls; dynamic turn limits prevent over-fission
Early exit / speculation	Speculative execution; guarded fast paths	Cascade early termination at confidence threshold; dynamic turn limits
Overhead being amortized	Fork/join, barrier sync, SIMD setup + scalar remainder loop	Per-call fixed cost: system-prompt/context re-send, prefill, network round-trip, subagent cold-start re-derivation
Hard blockers (legality)	Loop-carried dependencies; aliasing; irregular memory	Sequential state dependencies between steps; tool side effects; safety gates that can't be skipped
Correctness model	Every variant is bit-identical-correct; profile optimizes pure cost	Variants produce different answers; dispatch is statistical, needs a quality proxy
Handling run-to-run variance	N/A - deterministic; profiles are exact counts	Profiles are distributions. Mitigations: batch-invariant kernels (bitwise determinism), self-consistency / majority vote, constrained decoding, consensus/debate
Redundancy elimination	CSE, PRE, loop-invariant code motion - compute once, reuse	Prefix/prompt caching (identical token prefix ⇒ mathematically identical KV, skip recompute), semantic caching (GPTCache-style: embed query, similarity-match, skip the call entirely), persistent artifacts
Rematerialization vs. spill	Recompute a value rather than spill/reload when recompute is cheaper than memory traffic	Recompute vs. fetch KV cache: short contexts favor recompute, long contexts (10k+ tokens) and larger models favor fetching offloaded cache; hybrid schedulers tune the recompute ratio
Hardware / target mapping	Map task to ISA, core type, accelerator; tune per CPU generation	Map agent → model → silicon: prefill (compute-bound) vs decode (bandwidth-bound) on different GPUs; MLIR-based agent-graph placement under end-to-end SLA
Feedback cadence	Mostly AOT (offline: instrument → rebuild); some JIT re-profiling	Both: online routing per-request (JIT-like) + offline topology/prompt optimization (AOT-like)
Tooling / frameworks	GCC -fprofile-use, LLVM PGO, AutoFDO, BOLT	RouteLLM, FrugalGPT, AutoMix, Route-to-Reason, Router-R1, DSPy, MALBO, AgentOpt
Auditability of decisions	Deterministic, reproducible from profile + source	Signed route receipts: cryptographically attested per-decision records for attribution / billing / safety non-repudiation
Characteristic failure mode	Stale profile → mis-optimized hot path; profile not representative of real input	Cascade mis-escalation under adversarial input; over-fusion degrading quality; mis-calibrated stop judge

Appendix

A1. Context Inheritance

Subagents do *not* inherit the parent's context. Each one starts cold. It gets only the prompt the parent writes for it, runs in its own separate context window, and returns just its final message back to the parent. The intermediate work (its own file reads, reasoning, dead ends) is discarded. That's a deliberate trade-off, not an oversight:

- *Benefit*: the subagent's noise never pollutes the parent's context. A search that reads 40 files returns a three-line answer; the parent stays lean. This is the main reason to spawn at all.
- *Cost*: If the parent under-specifies the prompt, the subagent re-derives context from scratch — re-reading files, re-discovering structure — burning tokens and time.

So the "inheritance" is manual and lossy by design: the parent must hand-write a self-contained brief.

A2. Persistent Artifacts

For multi-agent workflows, persistent artifacts such as the following are maintained outside the context window.

- Repository files
- Execution logs
- Intermediate plans
- Scratchpads
- Generated summaries
- Search indices
- Vector embeddings
- Structured state

The model only loads what is needed for the current reasoning step.

A3. Overheads

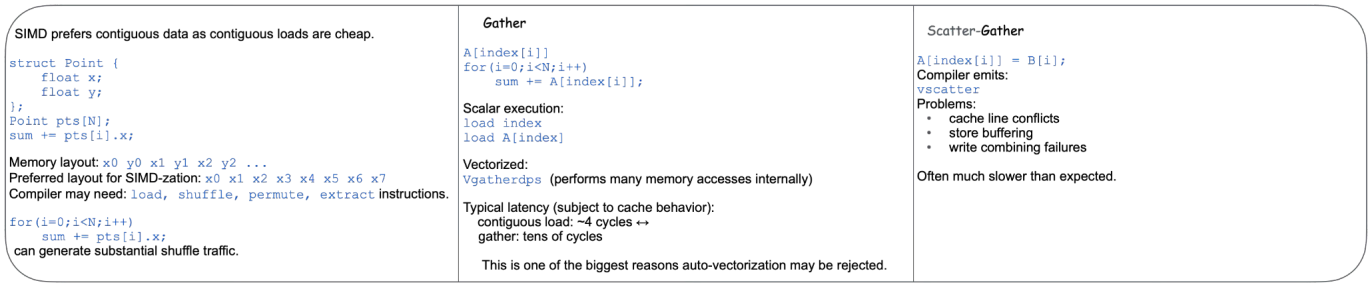
- *OpenMP (threads):* fork/join, barrier synchronization, cache-coherence traffic, and - the dominant ceiling in practice - memory bandwidth. Threads share one memory subsystem, so a bandwidth-bound loop hits a wall no matter how many cores you throw at it. Consider:

  ```
  #pragma omp parallel for
  for(i=0;i<1000;i++)
    ...
  ```

  The runtime must: (i) create or wake threads, (ii) distribute iterations and (iii) synchronize at loop end. For a small trip count: Useful work < scheduling overhead. Sequential execution wins. Compiler cost models therefore frequently reject OpenMP parallelization for short loops.

- *MPI (processes):* all of the above plus explicit message-passing latency and data serialization - worthwhile only at coarse granularity (large per-rank work, low communication-to-computation ratio).

- *Vectorization (SIMD):* loop-prologue/epilogue setup, the scalar remainder loop, and expensive gather/scatter when memory access is non-contiguous.



  LLVM's vectorizer documentation and ongoing LLVM discussions show that vectorization decisions are governed by profitability heuristics because SIMD can introduce additional overhead.

- On the agentic execution front, a coding task, for example, pays the following overhead:
  - task decomposition
  - context creation
  - retrieval
  - coordination
  - merging
  - conflict resolution

A4. Preference Data

It refers to pairwise judgments of which answer is better - not labeled "correct" answers, but comparisons. Concretely, for a query you have two responses (say from a strong and a weak model, or from human/LLM judges), and a label saying which one won, or whether the gap was meaningful. RouteLLM trains on exactly this kind of data - Chatbot-Arena-style "A beat B on this prompt" battles. The router learns the pattern "queries that look like this are ones where the strong model actually wins; queries that look like that get answered just as well by the weak model." The point is that the training signal is relative quality, which is what you can actually collect at scale (humans/judges are far better at "which is better" than at scoring absolute quality), and it's precisely the signal a router needs: not "how good is this answer" but "is the expensive path worth it for this input."

A5. Minimizing output variance

Primarily, there are three approaches - from bitwise to semantic - to minimize output variance.

- *Bitwise determinism (numerics):* Thinking Machines Lab's *Defeating Nondeterminism in LLM Inference* shows the culprit for T=0 non-reproducibility is lack of batch invariance: RMSNorm, matmul, and attention kernels change reduction order with batch size / sequence slicing, so server load alters logits and flips greedy decoding. Replacing these with batch-invariant kernels yields 1000/1000 identical runs — at ~60% throughput cost. SGLang/LMSYS did the same for reproducible RL training.
- *Semantic stabilization (aggregation):* Even under deterministic decoding, reasoning is unstable, so sample-and-aggregate: **self-consistency** (majority vote over N samples at T≈0.5–0.7, N≈5–20) reduces *variance, not bias*; **Universal Self-Consistency** uses the LLM as a semantic judge for long-answer tasks where exact-match voting fails (~10% extra inference); latent / representation-space self-consistency, semantic entropy, and Minimum-Bayes-Risk selection generalize this.
- *Format/structural determinism (constrained decoding):* Grammar-guided generation and token masking force valid JSON/SQL/regex output — SynCode reports ~96% fewer syntax errors, type-constrained decoding >50% fewer compile errors, XGrammar up to 100× faster than prior constrained decoders. Hybrid "reason-then-constrain" frameworks let the model think freely, then lock format.