

Modern AI systems: agents and system optimizations

Agent design V: cost-effective agents

Juncheng Yang



Harvard John A. Paulson
School of Engineering
and Applied Sciences



HARVARD UNIVERSITY

MaDSys

Measurement and Design of Systems Lab

Logistics

- Assignment 3 starting today
 - Much harder than earlier assignments
 - Leaderboard: limited submissions per day, no late days
 - Worth 20% of your grade: treat it like a midterm
 - Start early
 - Office hours next week (time TBA)
- Assignment update
 - Assignment 4 is now worth 20 points, and Assignment 5 is worth 10 points
 - Reason: Assignment 5 was too hard and overlapped with the course project and other courses' projects

Logistics

- Schedule update
 - Some lectures move earlier to fit the assignment changes
- Two bonus paper-presentation lectures: sign-up opens today
- Claude Code session log
 - Update the logs for all related sessions
 - **Do NOT alter session logs: it violates academic integrity**

Plan for today

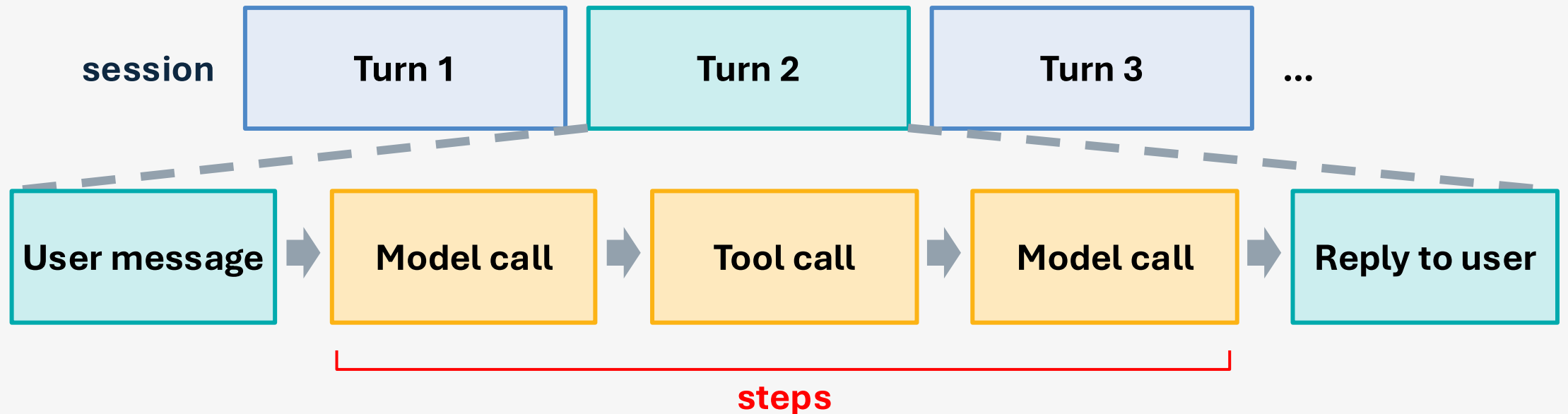
- Cost-effective agents
- Agent architecture
- Coding agent case study

Design Cost-Effective Agents

This lecture focuses on LLM cost while ignoring other compute costs

Recall the structure of a session

one user **turn** = multiple LLM and tool **steps**



Where does cost come from?

$$\text{Cost} = \text{Users} \times \text{Sessions / User} \times \text{Turns / Session} \times \text{Steps / Turn} \times \text{Tokens / Step} \times \text{Price / Token}$$

↑ **GROW**
adoption and engagement

↓ **SHRINK**
agent trajectories

↓ **SHRINK**
unit price

Your Claude Code usage: turns per session

- Most sessions have only one prompt
 - Good: most of you start a new session for each task
- A few run marathon sessions

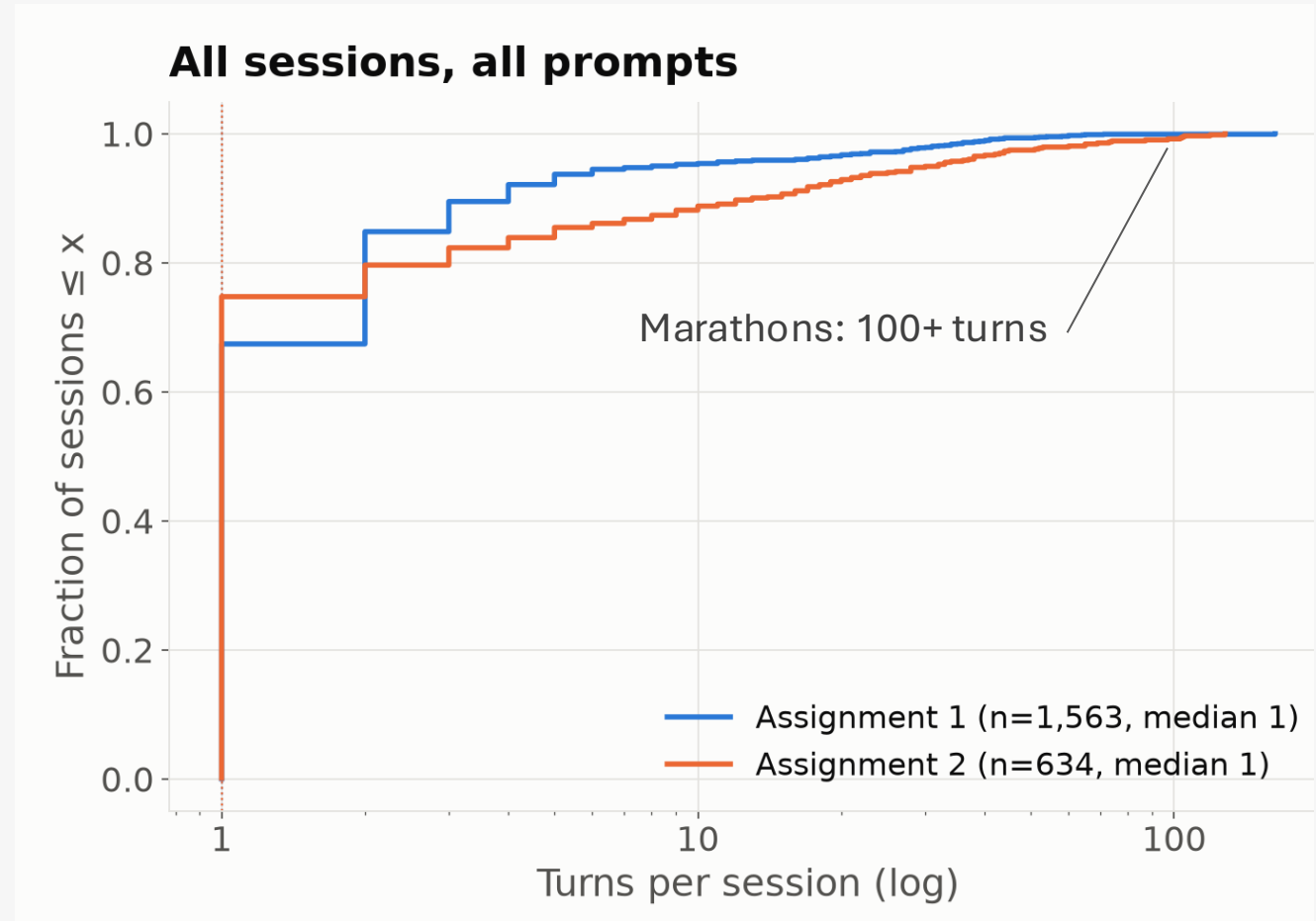
Single-prompt sessions

68%

Assignment 1

75%

Assignment 2



Reduce (user) turns per task

Turn k re-reads turns $1 \dots k-1$, so total cost grows with the square of turn count

Why the user comes back: no plan, stopped early or never checked

Front-load planning

Confirm goal and plan once, before acting

Structured task decomposition

Split into subtasks, each with a done check

Self-verification before handoff

Run tests and check output before returning

Goal: finish the task autonomously, with no user check-ins

Your Claude Code usage: steps per turn

- Most turns take few steps
 - Mostly because effort is set to medium or high
- Some turns run almost 1,000 steps

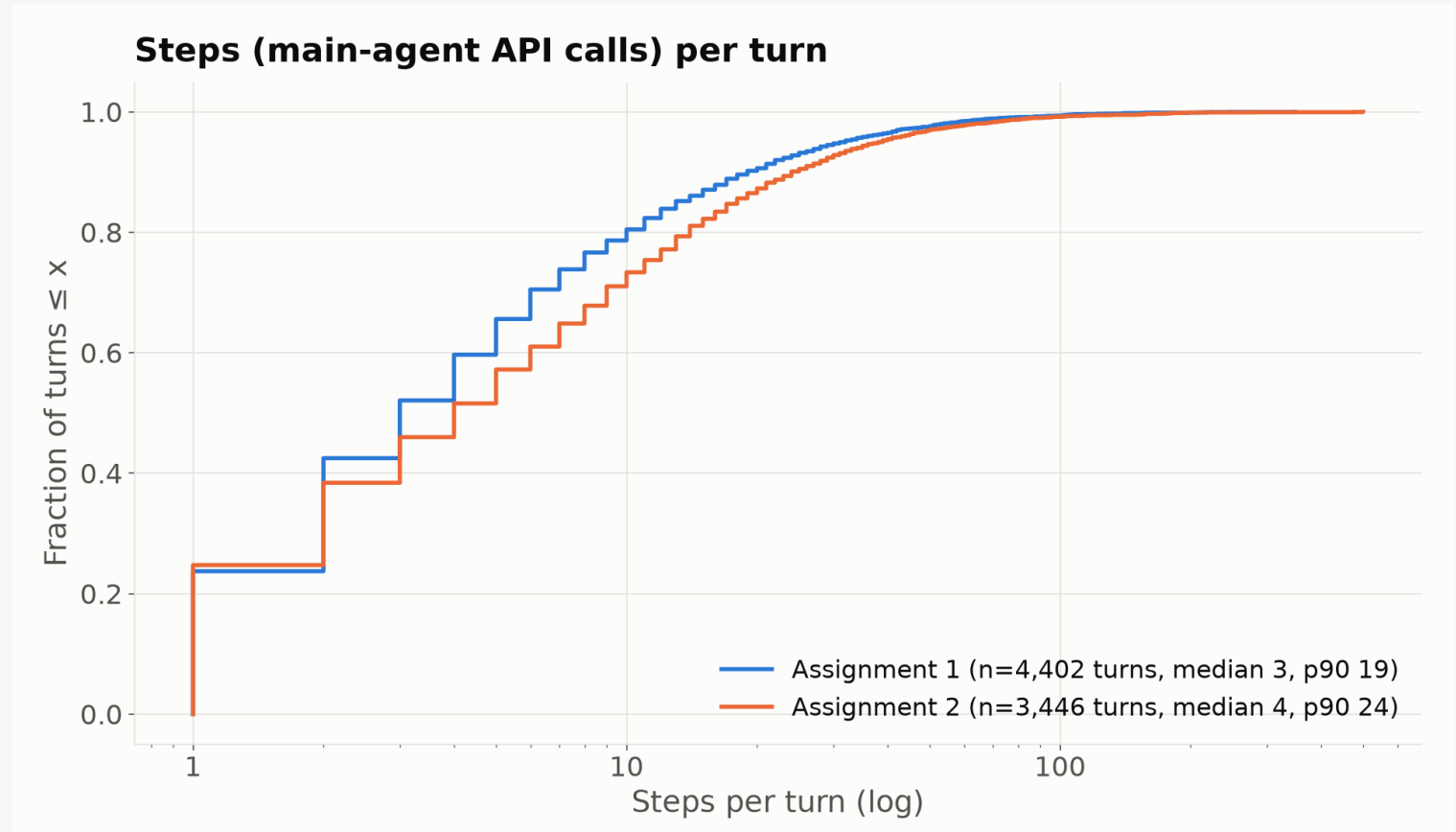
Median steps per turn

3

Assignment 1

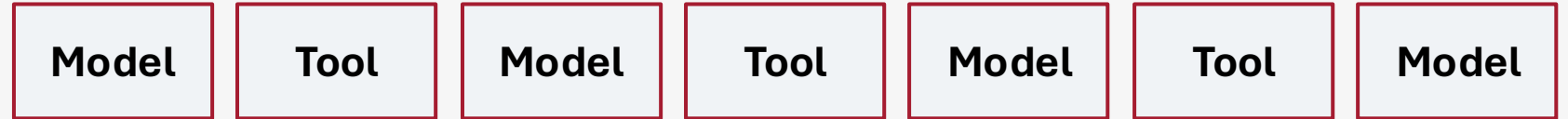
4

Assignment 2



Reduce steps per turn

One call per action
4 model calls



Plan, then batch
2 model calls



Batch independent calls

Read five files in one step
instead of five

Reuse plans and skills

Replay what worked before
instead of re-deriving it

Load context up front

Read the files you need at the
start, not one per step

Every model call re-reads the full context, so fewer calls cost less



Reuse plans, skills and experience: related papers

Skill libraries

Voyager (2023): stores executable skills

SkillRL (2026): improves skill library and agent policy jointly

Insights from experience

ExpeL (2023): extracts reusable insights

ReasoningBank (2025): keeps strategies from successful and failed runs

Playbooks and plan templates

Test-Time Plan Caching (2025): reuses and adapts plan templates

Durable conversational memory

Mem0 (2025): stores and updates facts

LightMem (2025): consolidates offline

SimpleMem (2026): compresses memory

Reuse pays when tasks recur; memory upkeep and stale entries can erase the savings.

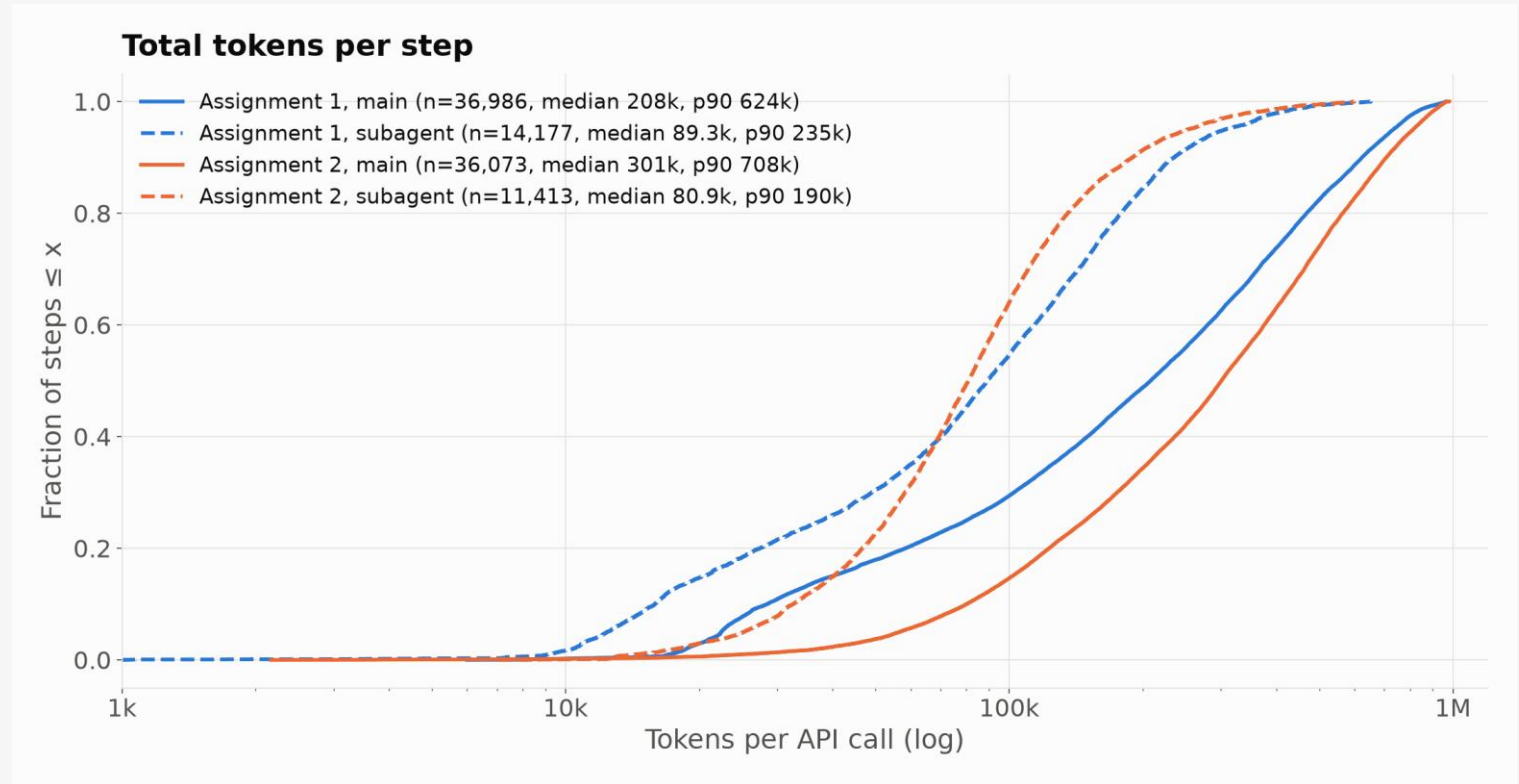
Your Claude Code usage: tokens per step

- Main agent
 - Median 200K–300K
 - p90: 600K–700K
- Subagent
 - Median ~80K
 - p90: ~200K

Main-agent median

208K **301K**

Assignment 1 Assignment 2



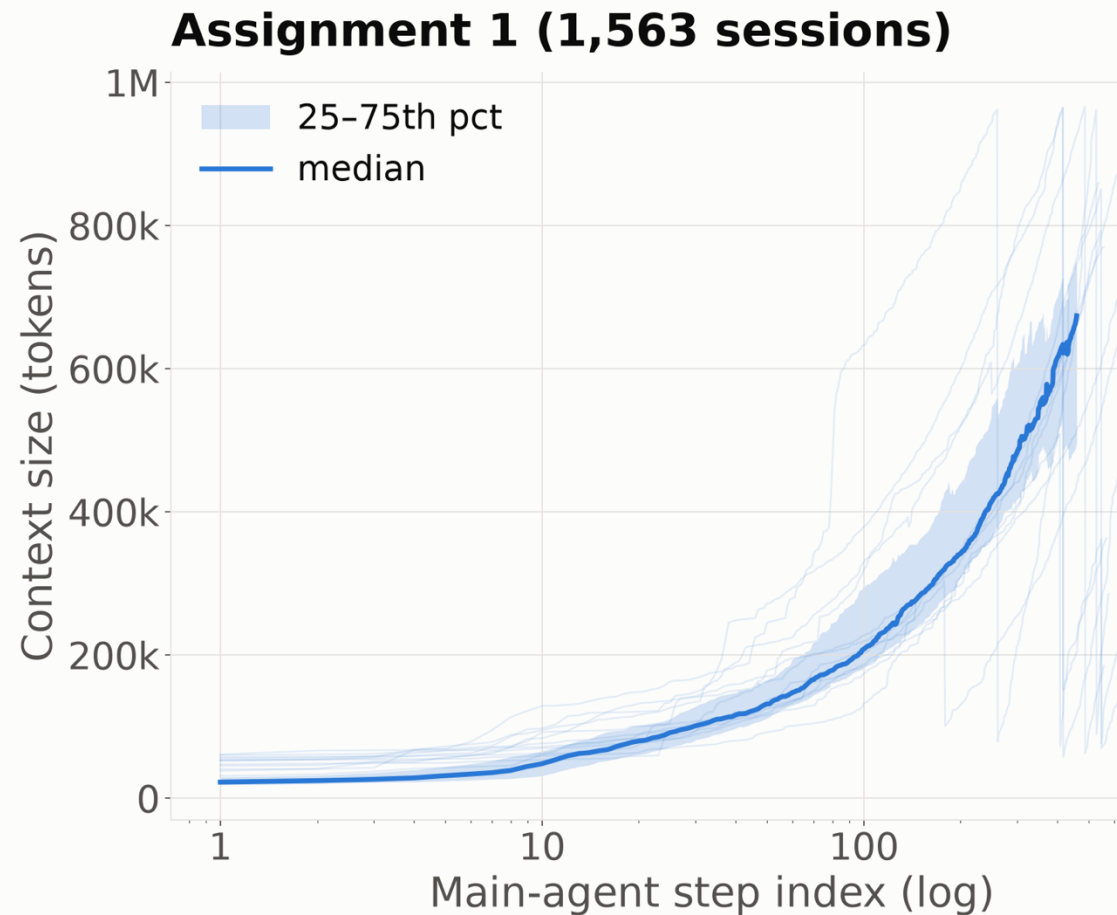
Your Claude Code usage: tokens over steps

- Context accumulates and grows quickly

Context at step ~500

670K **650K**

Assignment 1 Assignment 2



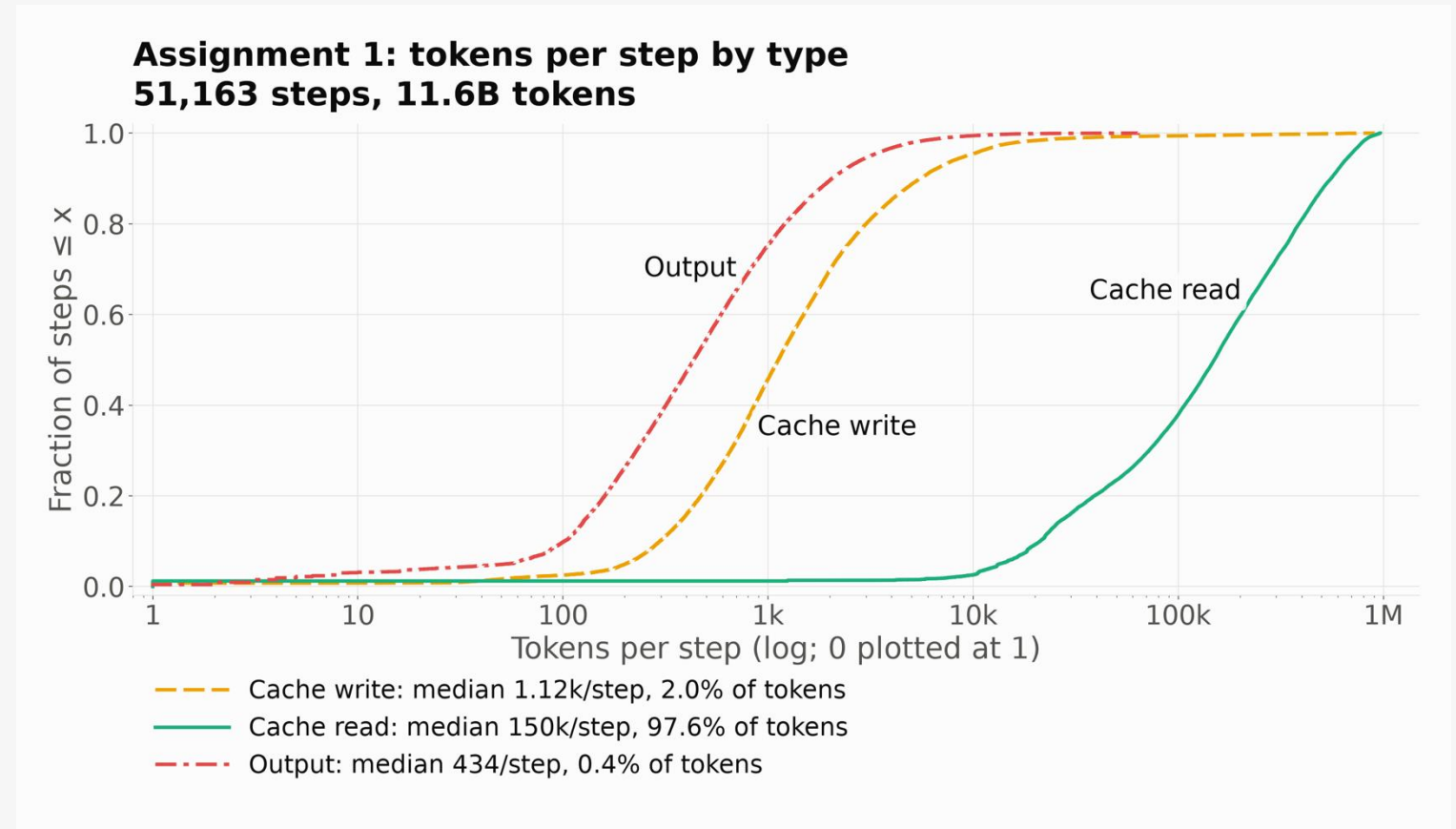
Your Claude Code usage: most tokens are cache read

- Output is 5× smaller than fresh prefill (cache write)
- Fresh prefill is 100× smaller than cache reads
- Typical price ratio, cache read : input : output = 1 : 10 : 50

Cache-read token share

97.6%

Assignment 1



Reduce input tokens per step: keep context lean

Drop what's stale	Truncate stale results Mask tool outputs already used	Compact the history Summarize old turns into state
Shrink what enters	Compress observations Keep errors, not the full log	Offload to files Save big outputs; pass a path
Load only what's needed	Retrieve selectively Only relevant tools and memory	Isolate exploration Sub-agent returns a summary

Tokens that stay in context are paid for again on every later step



Reduce input tokens per step: related papers

Compress agent history

ACON (2025): peak context cut 26–54%

Active Context Compression (2026):
agent compacts itself, 23% fewer tokens

TRACE (2026): summaries cause rework

Prompt compression

LLMLingua (2023): up to 20× fewer tokens

LongLLMLingua, LLMLingua-2 (2024)

Prompt Compression in the Wild (2026):
pays off only on long prompts

Context as external memory

MemGPT (2023): context as fast memory

Scroll (2026): lossless external log;
code picks what the model sees

Keep tool data out of context

Code execution with MCP (2025):
filter results in code, 150K → 2K tokens

Tool Search (2025): 85% fewer tool tokens

Programmatic Tool Calling: 37% fewer

Reduce output tokens per step: ask for less

Prompt-level constraints

State length, format and what to skip

```
Answer in ≤100 words.  
Return only JSON.  
One sentence. Top 3 items.  
Don't restate the problem.  
Don't explain your reasoning.
```

Terse system prompt: “Be concise; omit background unless requested.”

Output-token caps

Set a hard limit in the API call

```
max_tokens=300  
max_output_tokens=300
```

- Simple, but **truncation can cut an answer mid-sentence**
- Pair it with a concise-output prompt
- **Catch: reasoning can use up the cap**

Reduce output tokens per step: schema and effort

Structured output

Require a schema with only needed fields

```
{"action": "search",  
  "argument": "KV cache"}
```

- **Restrict enums and optional fields** so the model cannot add extras
- Suits tool-using agents: **prose around a tool call is pure cost**

Adaptive reasoning effort

Match the thinking budget to the step

```
lookup: effort="low"  
debug:  effort="high"
```

- Low effort for lookups and formatting; **high for planning and debugging**
- Set per call or per step, as **reasoning effort or a thinking-token budget**



Reduce output tokens per step: related papers

Prompts and budgets

Concise CoT (2024): ask for brevity

CCoT (2024): word limits in CoT

Chain of Draft (2025): terse drafts

TALE (2024): reasoning-token budget

Adaptive reasoning effort

Ares (2026): effort chosen per step

When to Plan (2025): plan only if useful

s1, L1 (2025): enforce thinking budgets

Reason Efficiently, ShorterBetter

Structured output and decoding

Let Me Speak Freely? (2024): strict formats can hurt reasoning

XGrammar, SGLang (2024)

JSONSchemaBench (2025)

Emit only what changes

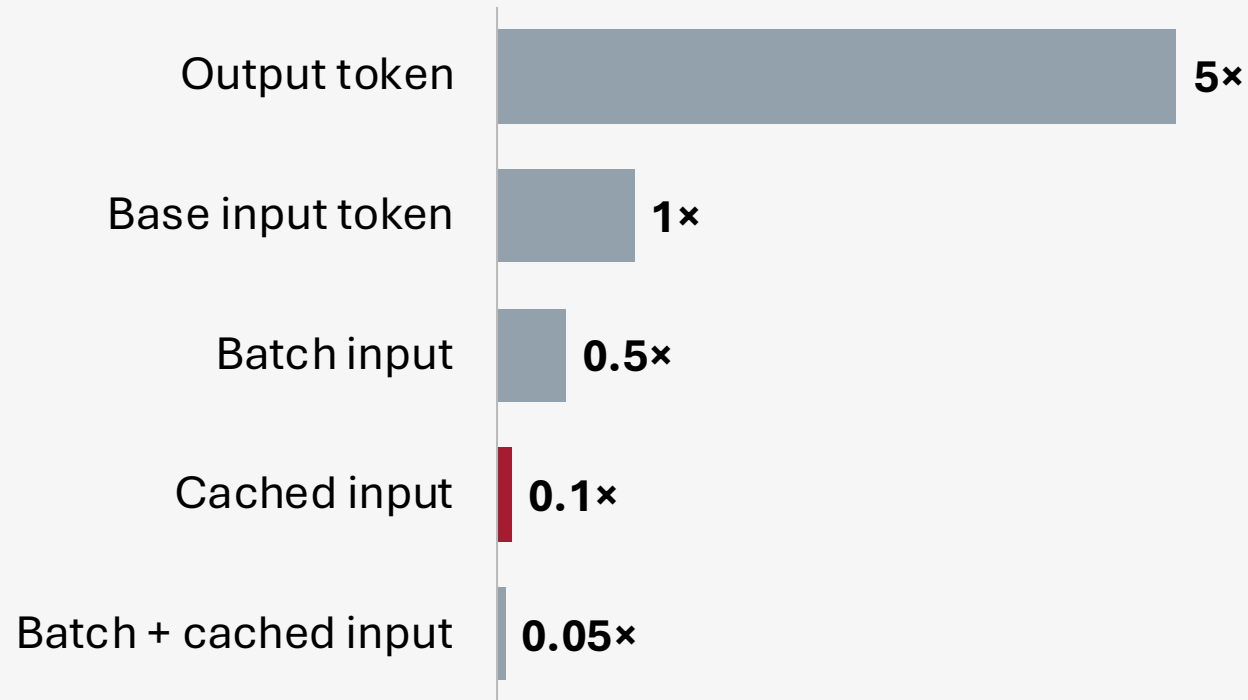
AdaEdit (2026): diff or whole file

CodeStruct (2026): AST-level edits

Tool-making agents (2026): reusable tools cut output tokens 58%

Reduce price per token: caching and batching

Price per token relative to base input, Claude API



Maintain stable prefixes

system prompt, tools, early history

Batch what can wait

evals, backfills, bulk jobs

Reduce price per token: model routing

Choose up front

Pick one model per turn, before generating

How: a router trained on preference data; can also pick agent roles

Escalate on failure

Try a cheap model first; move up when a check fails

Checks: a learned scorer, self-verification, or execution feedback

Re-route each step

Re-select models as the agent's trajectory grows

How: route on history and budget; an orchestrator calls models as tools

Hard part: one cheap planning mistake can waste many later actions

Reduce price per token: system techniques in later lectures

Model optimization

- MoE / sparsity
- Quantization
- Pruning
- Distillation
- Workload-targeted fine-tuning

Serving engine optimization

- Speculative decoding
- KV-cache optimization
- Better request scheduling

Hardware

- Faster kernels
- Better kernel scheduling
- Specialized accelerators



Reduce price per token: related papers

Prefix caching

vLLM (2023): PagedAttention

SGLang (2024): RadixAttention prefix cache

Prompt Cache (2024): modular reuse

Choose up front

RouteLLM (2024): learned router

MasRouter (2025): also picks agent roles

Escalate on failure

FrugalGPT (2023): learned cascade

AutoMix (2023): self-verification

EcoAssistant (2023): execution feedback

Re-route each step

ToolOrchestra (2025): small orchestrator

BAAR, MTRouter (2026): budget-aware

Every saving has a counter-cost

Compaction vs. reconstruction

Dropped files get re-read, and the summary rewrites the cache

Routing vs. longer failures

One bad plan from a cheap model can add many steps or a failed attempt

Subagents vs. duplicated context

Each subagent re-reads the shared files and instructions

Speculation vs. wasted work

Wrong guesses burn tokens and contend for the same tools

Net saving = gross saving – counter-cost; measure both on the same tasks

Cap each task's budget and stop when progress stalls

Per-task limit	Example
Dollars	\$10
Model calls	80 calls
Tool time	15 min
Retries per step	2 retries

Set limits from traced runs, e.g. 2× the p95 of successful tasks

Stop when

- The verifier passes
- No new passing test in N steps
- The same error repeats
- A limit hits: hand off with a summary

A stopped task with a summary costs less than a runaway one

Example limits are illustrative.

Do you need an LLM at all?

Same steps every time?	 Write a script Have the LLM write the code once, then just run it
Fixed input format?	 Parse with rules Regex, parsers and templates cost nothing per call
Narrow label or score?	 Train a small classifier GBM or Jev!
Fixed workflow, little reasoning?	 Script the workflow Call the LLM only at the steps that need it

Reach for an LLM only when a step needs reasoning

Make Your Agent Faster

Faster agents are usually, but not always, cheaper too

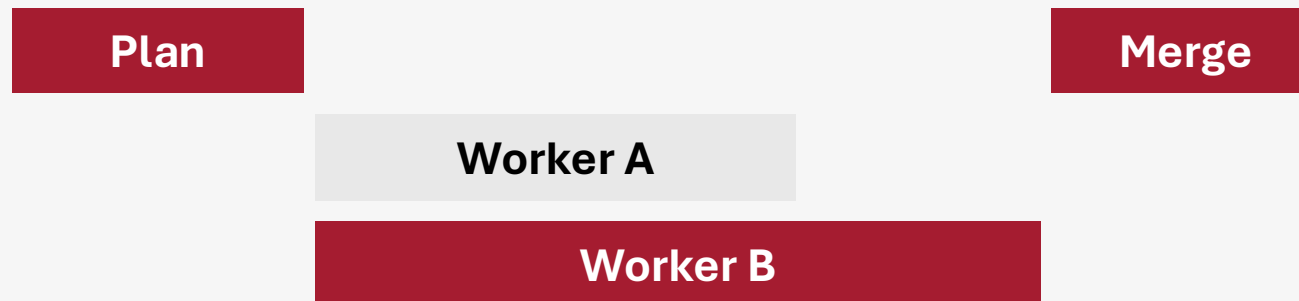
- Speed up LLM serving
- Improve orchestration
 - Speed up tool calls
 - Parallelize work
- Speculative actions

Completion time is the critical path, not the sum of every step

Serial: time $\approx$ sum of every step



Parallel: time $\approx$ critical path



Cheaper can be slower

Small models may take more steps

Parallel costs more

Less waiting, more total tokens

Red: the critical path that sets completion time

Speed up LLM serving

Every cost technique from the previous section also cuts latency



Fewer turns



Fewer steps



Fewer tokens



**Cheaper
serving**

Caching, smaller
models

Improve orchestration



Async execution

e.g., start the next step while a tool call runs



Persistent sessions

e.g., keep one shell or browser open across steps



Stream and pipeline

e.g., start the tool call before the LLM response finishes



Colocate services

e.g., run tools on the same host as the model

Speed up tool calls

Builds, tests and page loads can take longer than the model call itself

Cache tool results

Repeat reads and searches hit the cache

Faster hardware

Run heavy compute on a server, not a laptop

Better tools

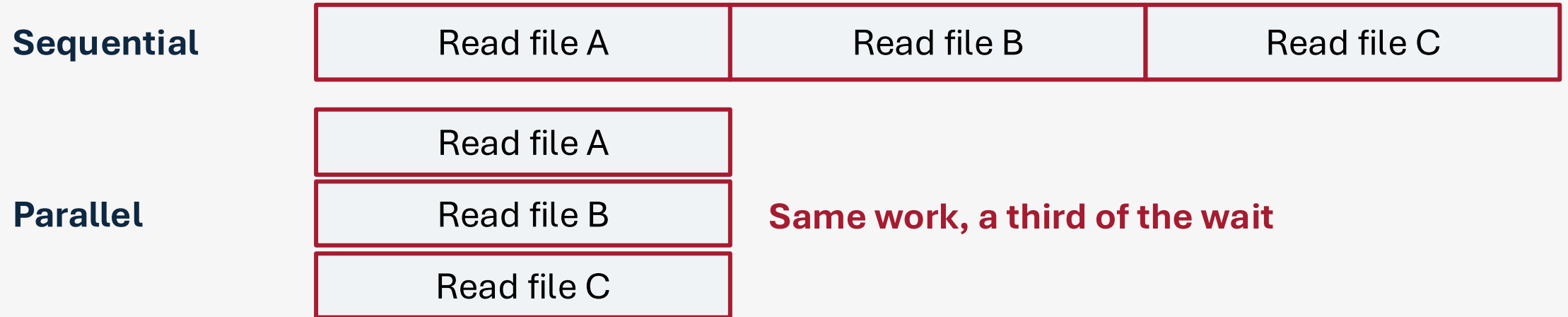
Faster, cheaper tools for the same job; call tools selectively

Fail fast

Time out hung calls

Measure tool time per step before tuning the model

Parallelize work



Parallel tool calls

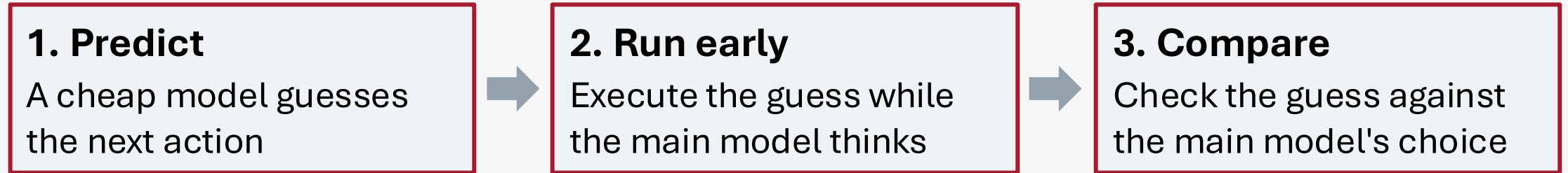
Issue independent calls in one step

Parallel sub-agents

Split independent subtasks; merge the summaries

Only independent work runs in parallel; shared state needs a merge step

Speculative actions



Guess matches

The result is ready; the wait is skipped

Guess misses

Discard it; the cost is one wasted call

Pays off when guesses are often right and actions can be undone

Open question

How to schedule speculative calls under limited GPU, CPU and API budgets so the critical-path time saved exceeds the work wasted



What gets speculated

The horizon grows from one call to a whole trajectory

Tool intent

Which tool comes next
SPORK [8]

Tool call

The tool plus its arguments
*PASTE [5],
Speculative
Tool Calls [2]*

Next action

Any action or API response
*Speculative
Actions [1]*

Branches

Several possible paths at once
B-PASTE [6]

Trajectory

A multi-step chain, already executed
*Speculative
Macro Commit
[10]*

Sources: [1] Speculative Actions (ICLR 2026) · [2] Speculative Tool Calls (arXiv 2025) · [5] PASTE (arXiv 2026) · [6] B-PASTE (arXiv 2026) · [8] SPORK (arXiv 2026) · [10] Speculative Macro Commit (arXiv 2026)



Where predictions come from

Model-based

A small draft model predicts what the large model will do next

*Speculative Actions [1],
Speculative Tool Calls [2]*

Self-speculative

An early fork of the same model predicts its own later tool call

SPORK [8]

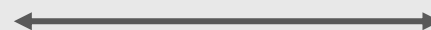
History-based

Past traces predict the next tool, and its arguments bind to the last result

PASTE [5], Speculate with Memory [9]

Lossless

Always verify before keeping work



Lossy

SPAgent [3] skips some checks

Sources: [1] Speculative Actions (ICLR 2026) · [2] Speculative Tool Calls (arXiv 2025) · [3] SPAgent (arXiv 2025) · [5] PASTE (arXiv 2026) · [8] SPORK (arXiv 2026) · [9] Speculate with Memory (arXiv 2026)

Two examples

PASTE [5]

Predict the next tool from past traces

```
search(q)
  → visit(results[0].url)
  → extract(page)
```

Once search(q) starts, the trace pattern says a visit comes next. The URL is filled in from the search result, so the visit runs while the model is still generating.

SPORK [8]

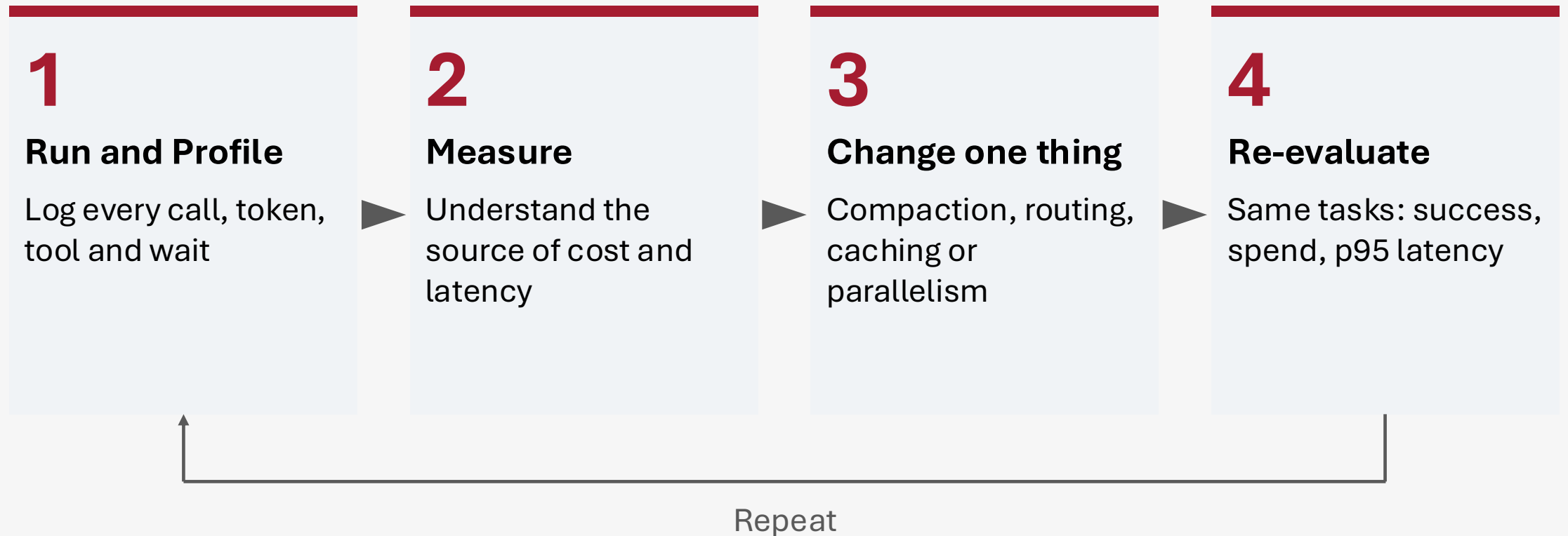
The model predicts its own tool call

```
think: 2,000 tokens ...
      early probe → search(q)
tool_call search(q): ready
```

The tool intent is often clear early in the reasoning. A forked probe launches the predicted call, and the result is waiting when the main model emits it.

Sources: [5] PASTE (arXiv 2026) · [8] SPORK (arXiv 2026)

Optimize in a loop: run, measure, change, again



Remember this for your Assignment 4!

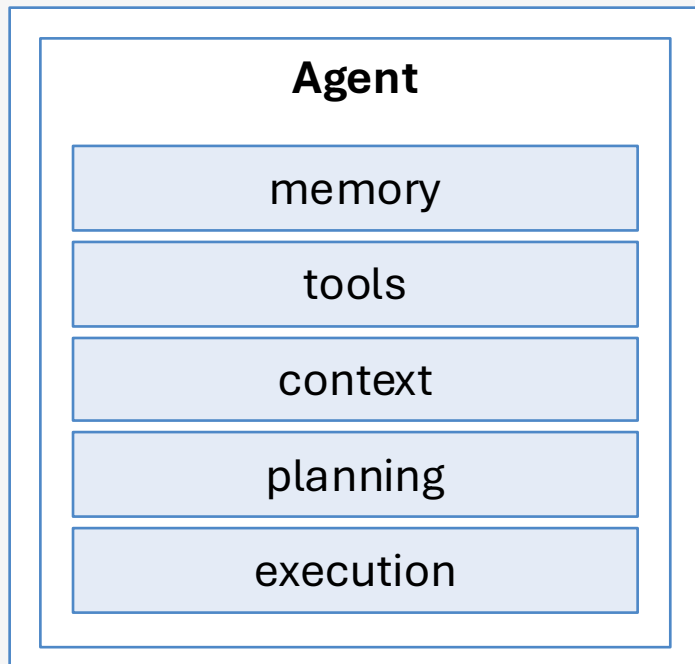
Agent Architecture

Monolithic · Modular · Microkernel

Three harness architectures

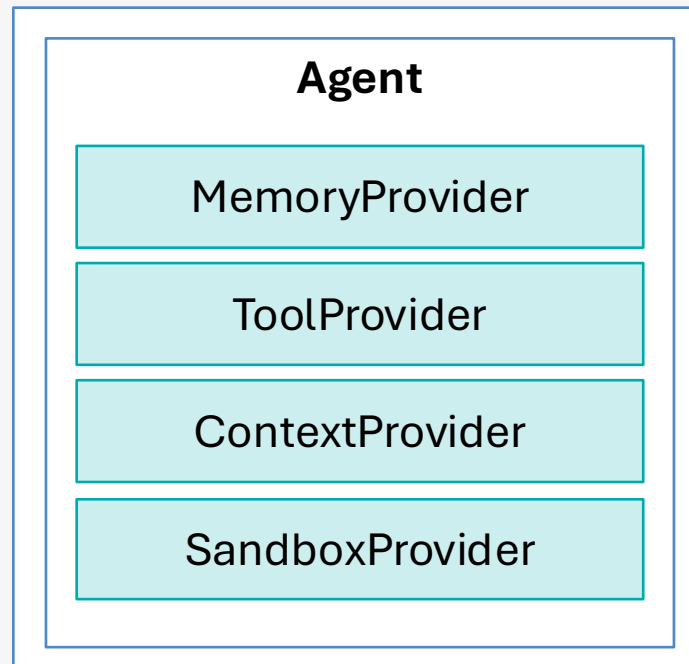
1 • Monolithic harness

Everything lives in one runtime



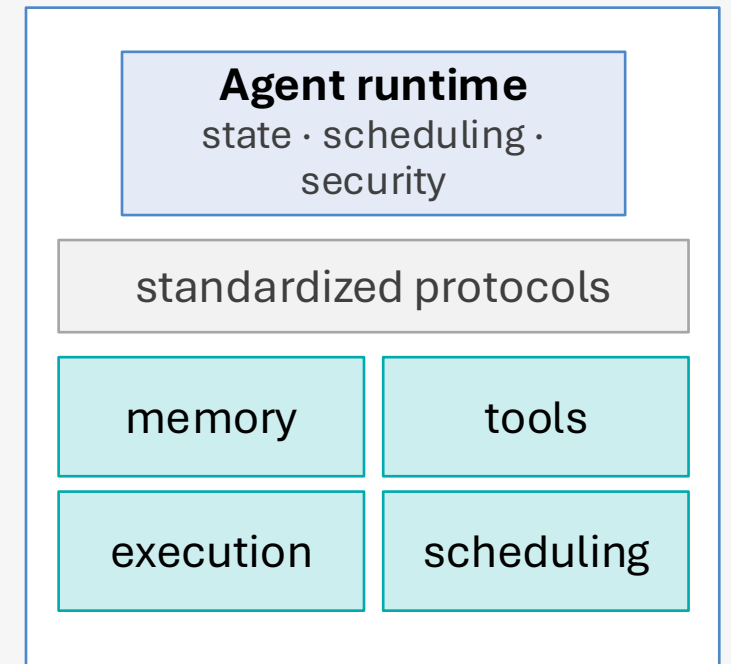
2 • Modular harness

Interfaces, one controller



3 • Agent microkernel

Core keeps coordination only



Shift: responsibility moves out of the agent, from code to interfaces to services.

Monolithic harness: one runtime implements memory, tools, context, planning and execution

```
class CodingAgent:
    def __init__(self):
        self.memory = Memory()
        self.tools = Tools()
        self.context = ContextManager()
        self.planner = Planner()
        self.executor = Executor()

    def run(self, task):
        while not self.finished:
            a = self.planner.plan(task)
            r = self.executor.execute(a)
            self.context.update(r)
```

Advantages

- Simple to implement and debug
- Minimal communication overhead
- Internal state is easy to share
- Low execution overhead

Disadvantages

- Hard to replace one component
- Components become tightly coupled
- Hard to share infra across agents
- Isolation is hard to enforce

Key characteristic: the agent owns everything, in one runtime.

Modular harness: the agent still controls the loop but reaches each component through an interface

```
agent = CodingAgent(  
    memory_provider=RedisMemory(),  
    tool_provider=MCPTools(),  
    context_provider=SlidingWindowContext(),  
    sandbox_provider=DockerSandbox())  
# Docker → Firecracker: swap one provider
```

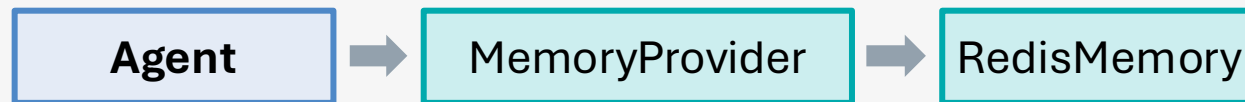
Advantages

- Easy component replacement
- Better testing and extensibility
- Clear separation of responsibilities
- Third-party implementations plug in

Monolithic



Modular

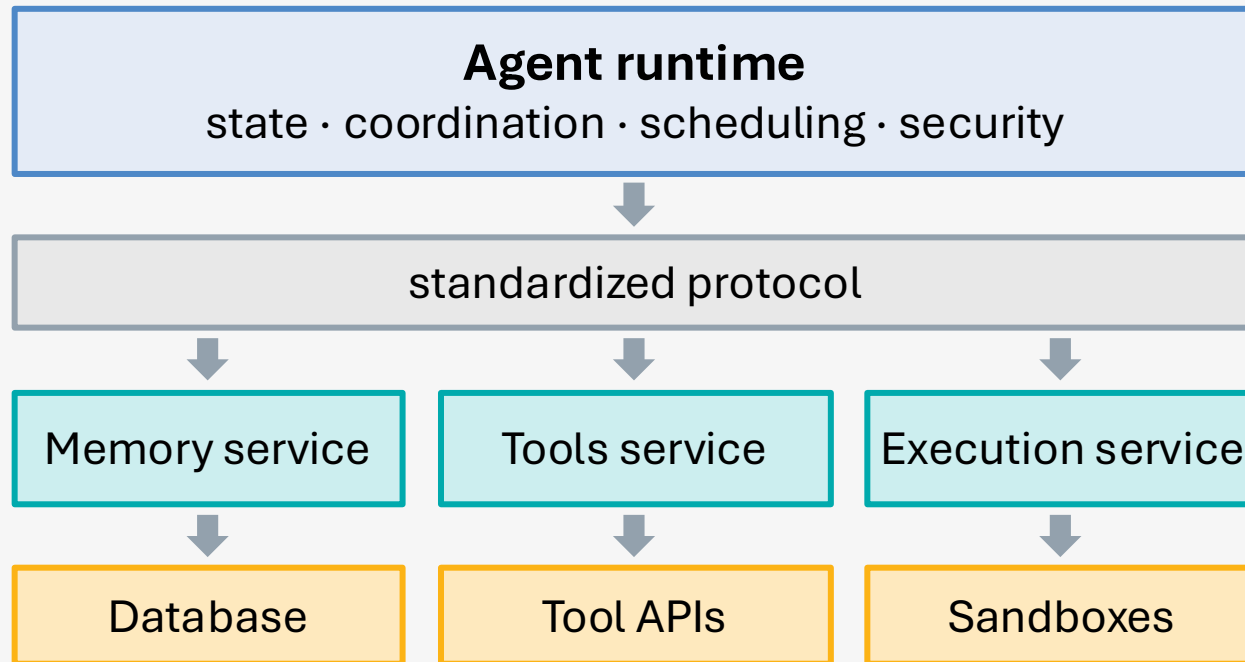


Disadvantages

- Controller still owns the lifecycle
- Coordination logic grows complex
- Failure domains can still be shared
- Joint tuning needs controller edits

Key characteristic: one controller delegates through interfaces.

Agent microkernel: a small runtime coordinates, and external services provide the capabilities



Services can run in separate containers or machines

Advantages

- Services scale independently
- Better fault isolation
- Stronger isolation, if enforced
- Shared, centrally managed infra

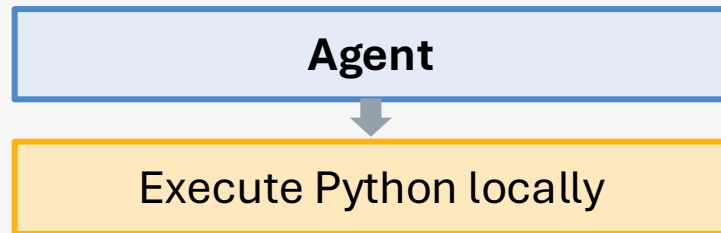
Disadvantages

- Higher communication overhead
- More failure modes to handle
- Harder end-to-end debugging
- Needs standard protocols, lifecycles

Key characteristic: the runtime coordinates; external services provide capabilities.

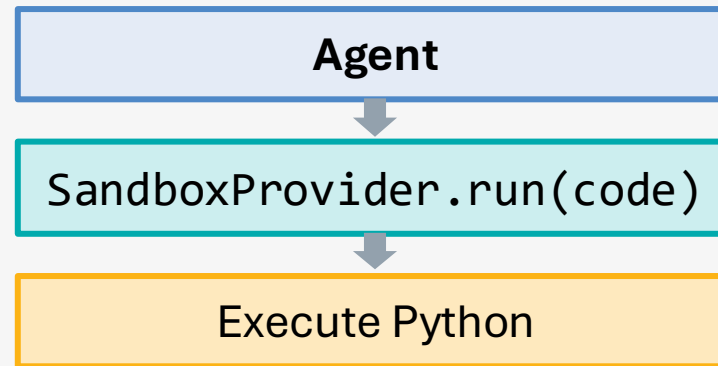
Running one Python snippet: only the microkernel hands sandbox management to its own service

1 • Monolithic



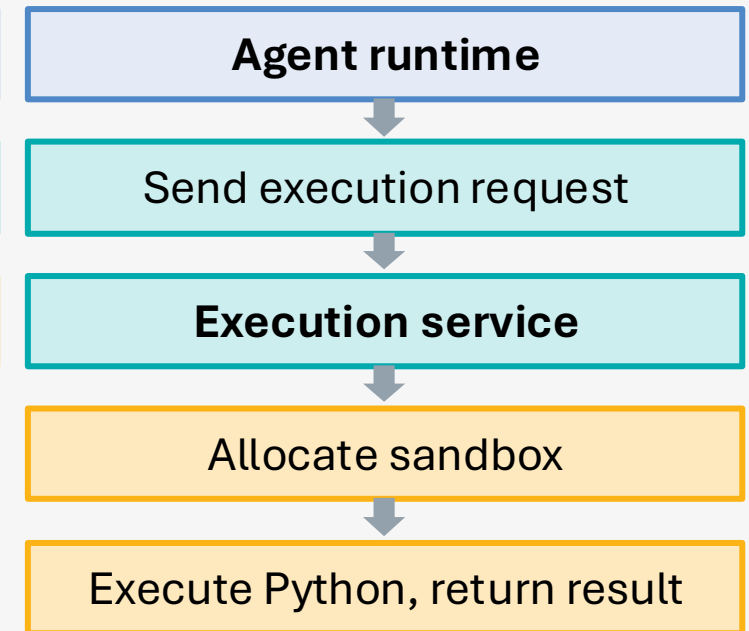
Fewest hops; isolation is hard to enforce

2 • Modular



Swap the provider, not the agent

3 • Microkernel



Microkernel: the execution service owns allocation, isolation and sandbox lifecycle.

The microkernel trades simplicity and low overhead for isolation, sharing and scale

Dimension	Monolithic	Modular	Microkernel
Design principle	Everything integrated	Interface abstraction	Minimal core + services
Central controller	Yes	Yes	Yes, but reduced
Component replacement	Difficult	Easy	Easy, if protocols match
Execution isolation	Optional	Optional	Architectural emphasis
Component deployment	Usually together	Together or separate	Often independent
Independent scaling	Limited	Possible	Strong
Communication cost	Low	Low–medium	Potentially higher
Typical use case	Standalone agents	Agent frameworks	Multi-agent infrastructure

Fit: monolith for standalone agents, microkernel for multi-agent infrastructure.

The eight harnesses

Claude Code

Anthropic · TypeScript
(closed)

Policy layer and typed
hooks around Claude

CC

Codex CLI

OpenAI · Rust

Sandbox first, ask
second

CX

DeepSeek Harness

DeepSeek AI · TypeScript

Everything is a plugin,
even the loop

DSH

OpenCode

Anomaly (SST) ·
TypeScript, Bun

A session server with
thin clients

OC

Pi

Mario Zechner / Earendil
· TypeScript

A small core you
extend in TypeScript

PI

oh-my-pi

Can Bölük / Stencil Labs
· TypeScript + Rust

Pi fork with native
tools and the IDE
wired in

OMP

OpenClaw

OpenClaw Foundation ·
TypeScript

An assistant gateway
for chat channels

CLAW

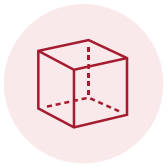
Hermes Agent

Nous Research · Python

An agent that writes
its own skills

HM

Unique to Codex CLI



Sandbox before approval

Commands run in a sandbox with no network access. The agent asks for approval only to go out of the sandbox.



Patches that obey a grammar

A Lark grammar sets the patch envelope. The agent matches hunks in four passes.

Unique to DeepSeek Harness



The loop is also a plugin

The loop, adapters, tools, log and approvals are plugins. YAML layers can replace each plugin.



Requests from the log

Each request is a frozen view of the session log. The harness only adds entries to this log.



Compare-and-swap edits

An edit fails if the file changed after the model last read it.

Unique to OpenCode



The harness is a server

The TUI, the SDKs and the web app use the same OpenAPI HTTP server.



Fuzzy edits with a size limit

Nine replacers go from exact to anchor match. The harness rejects fuzzy matches that are too large.



A snapshot at each step

A shadow git repository lets /undo restore the files, not only the chat.

Unique to Pi



A minimal core

The core has four tools and no MCP, sub-agents or approvals. Extensions supply all other functions.



Cache warming with a price check

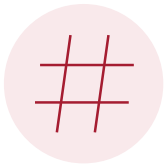
Pi sends keep-alive requests only when the expected saving is \$0.05 or more.



Libraries for other projects

pi-ai supports 42 providers on 10 wire APIs. Three other projects use Pi code.

Unique to oh-my-pi



Hashline edits

Each edit gives a 4-hex file tag and line numbers. The model does not type the old text again.



The IDE in the process

One shell stays open for all calls. Native tools give search, AST, LSP and a debugger.



Special context methods

snapcompact changes old history into images. TTSR stops and tries again when a regex matches.

Unique to OpenClaw



A chat gateway with queue lanes

27 chat channels send messages to one lane for each session. A queue mode controls messages that come during a run.



Memory before compaction

First, a silent turn writes dated notes. Recall uses full-text search and vector search together.



Fences, taint and sentinels

Fences isolate untrusted text. A network read taints the turn. The agent monitors secret sentinels.

Unique to Hermes Agent



A fork that learns

A background fork examines each nudged turn. Then it edits the memory and the skills.



Earliest compaction

Compaction starts at 50% of the window. This is the earliest of the eight harnesses.



Save before execution

The agent writes tool-call rows to the session database before any side effect occurs.

Next class

- Agent safety and security
- Self-evolving harness