Agent Tooling, Memory, and Context

CSE 291A · October 15, 2026

Yiying Zhang

Why LLM Agents stand out?

- **Language Mastery:** Their inherent capability to both comprehend and produce language ensures seamless user interaction.
- **Decision-making:** LLMs are equipped to reason and decide, making them adept at solving intricate issues.
- **Flexibility:** Their adaptability ensures they can be molded for diverse applications.
- **Collaborative Interactions:** They can collaborate with humans or other agents, paving the way for multifaceted interactions.





The "Brain and Limbs" Analogy

- LLM (The Brain):
  - Performs complex reasoning.
  - Understands user intent.
  - Formulates plans.
  - Summarize information and write output.
- Tools (The “Limbs"):
  - **Extend Capabilities**: Access data the brain doesn't have (e.g., Google Search).
  - **Improve Reliability**: Offload tasks the brain is bad at (e.g., use a calculator for math).
  - **Enable Real-World Actions** : Interact with the world (e.g., use an API to send an email).

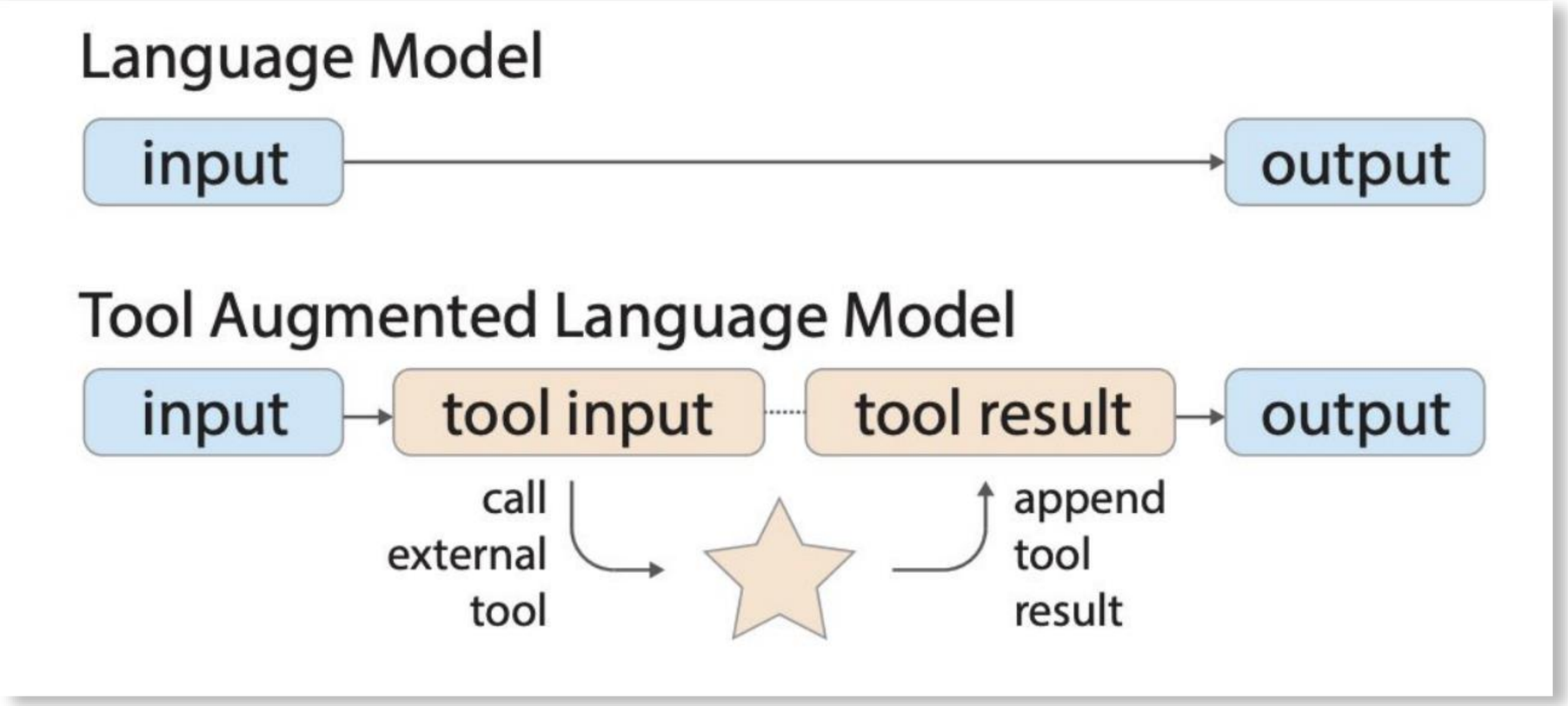
How Do Models Call Tools?

- LLMs do not execute tools directly.
- LLMs “learn” to generate tool call syntax
- A system orchestration layer (or simply your code) intercept and execute
- Three primary methods for LLMs to know about tools
  - Instruction Tuning (Fine-Tuning)
  - System Prompt (e.g., ReAct)
  - In-Context Learning (Few-Shot)

Method 1: Instruction Tuning

- An LLM explicitly fine-tuned on a dataset of (Instruction, Tool Call, Tool Output, Final Answer) examples, e.g., Toolformer, TALM
- Fine-tuned to recognize when an instruction requires a tool and to emit the *exact* syntax, like `<tool_call>weather_api(location='Seoul')</tool_call>`.
- Knowledge of tool use baked into model weights
  - **Pro:** Very fast and reliable for known tools.
  - **Con:** Not flexible; cannot easily add new tools without re-tuning.

LLMs + training for tool use: TALM



TALM: Tool Augmented Language Models

TALM: Tool Augmented Language Models

LLMs + training for tool use: Toolformer

The New England Journal of Medicine is a registered trademark of [QA("Who is the publisher of The New England Journal of Medicine?") → Massachusetts Medical Society] the MMS.

Out of 1400 participants, 400 (or [Calculator(400 / 1400) → 0.29] 29%) passed the test.

The name derives from "la tortuga", the Spanish word for [MT("tortuga") → turtle] turtle.

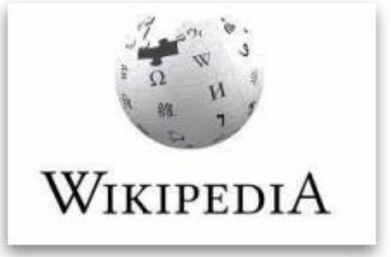
The Brown Act is California's law [WikiSearch("Brown Act") → The Ralph M. Brown Act is an act of the California State Legislature that guarantees the public's right to attend and participate in meetings of local legislative bodies.] that requires legislative bodies, like city councils, to hold their meetings open to the public.











Toolformer: Language Models Can Teach Themselves to Use Tools

Method 2: System Prompt

- A system prompt instructs the LLM to call tools, e.g., with a ReAct loop
- **Reason:** LLM thinks step-by-step about what it needs.
- **Tool Call:** LLM decides to call a tool and generates the tool call syntax.
- **Tool Execution:** Outside the LLM, execute the tool, get result, and feed back to LLM
- *Repeats* this loop until it has enough information to answer the user.
- **Pro:** Flexible; new tools can be added just by describing them in the prompt
- **Cons**: Can be less reliable than fine-tuning; agent can get "stuck in a loop”, making errors or inducing many LLM calls.

Method 3: In-Context Learning

- Provide a few examples (few-shot prompting)
  - `User: What's the weather in SF? Model: <tool_call>weather('SF')</tool_call> System: <tool_response>{"temp": 65}</tool_response> Model: The weather in SF is 65. --- (New query) --- User: What about Seoul?`
  - `Model: <tool_call>weather(‘Seoul')</tool_call>`
- **Pro:** Flexible; new tools can be added just by describing them in the prompt
- **Con:** Longer context, can be less reliable than fine-tuning, needs good examples

Common Agent Use Cases

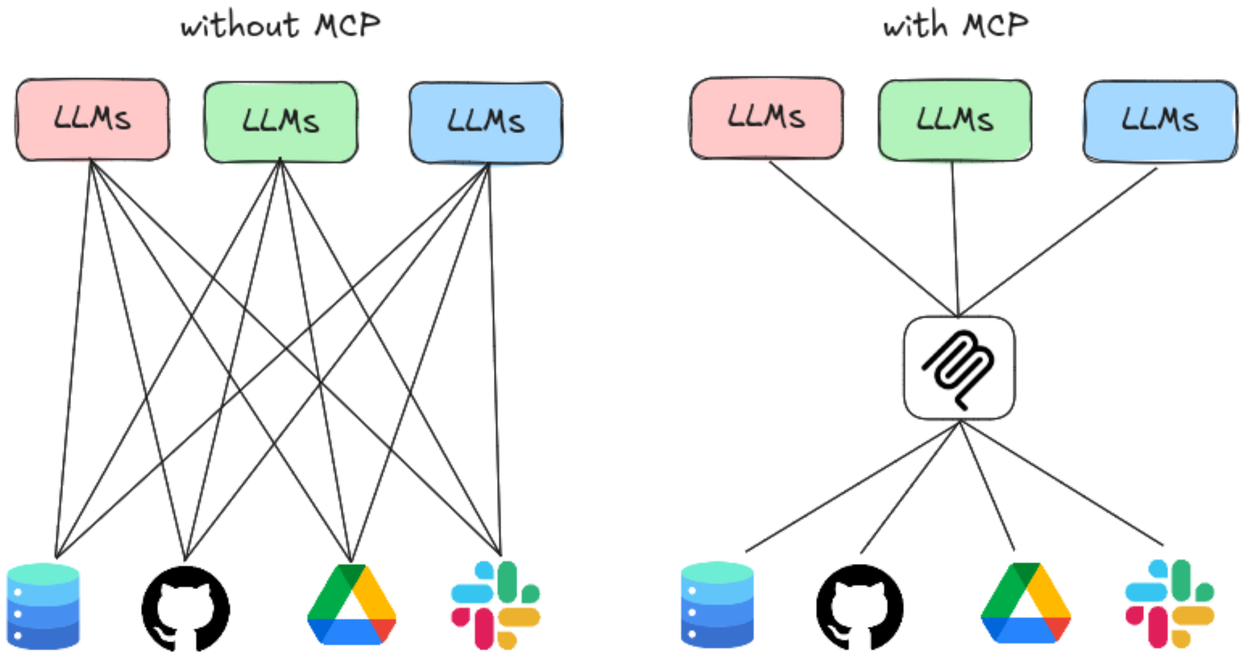
- **Data Science** : Writing and executing Python code to analyze data, plot charts, and check statistical results.
- **Search & Retrieval** : Answering questions with up-to-date information by fetching and summarizing web content.
- **Productivity** : Managing calendars, drafting emails, and summarizing documents.
- **DevOps**: Reading logs, diagnosing errors, suggesting code fixes, or managing cloud infrastructure via CLI tools.
- **Creative Work**: Generating images with tools like DALL-E, or composing music snippets via a MIDI API.

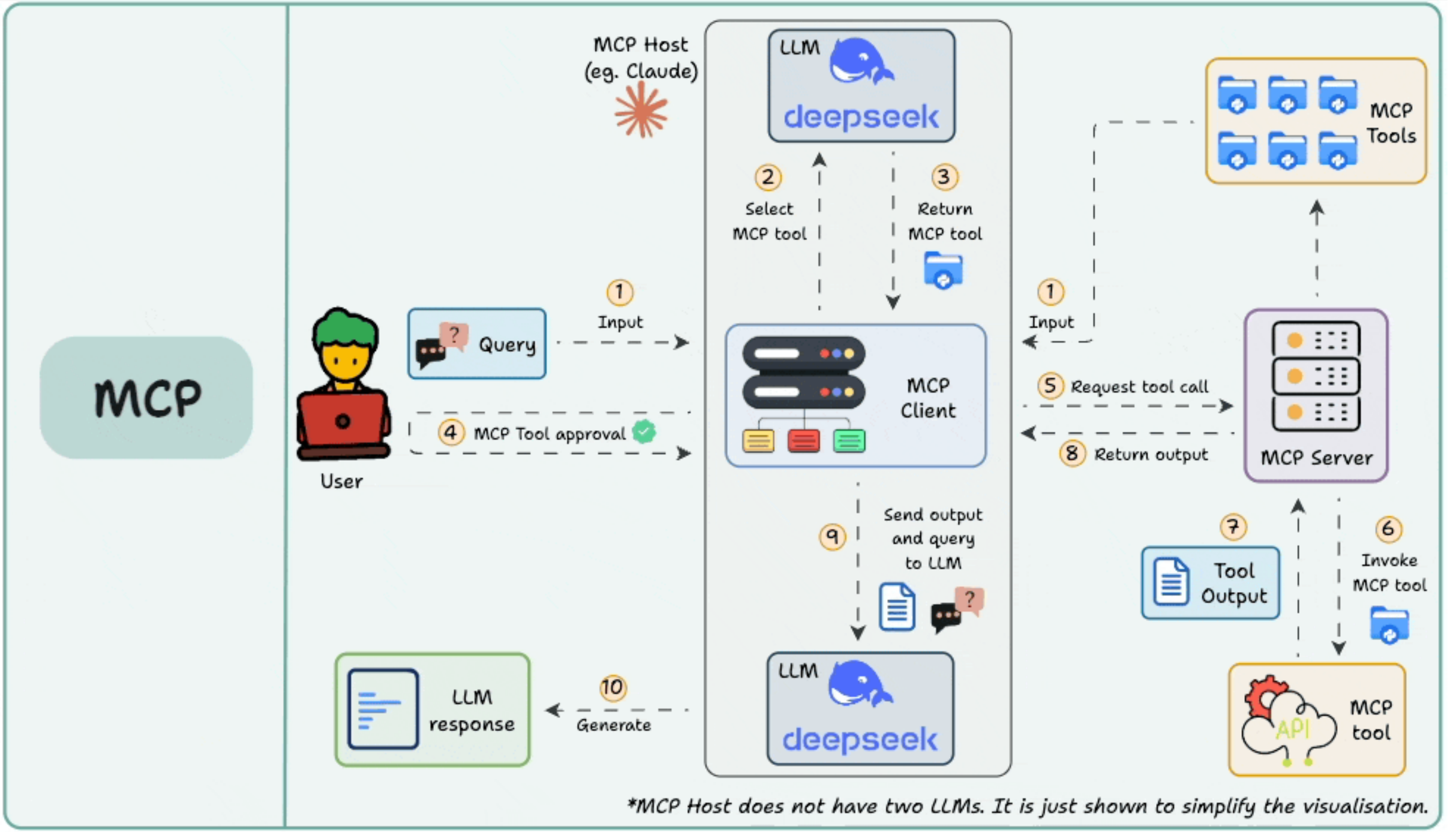
Agent Tooling Challenge 1: The Standardization Problem

- **The Problem:** An "N x M" integration nightmare.
- **N Models:** (GPT-4, Claude 3, Gemini, Llama 3...) all have different fine-tuning data and preferred tool-call syntax.
- **M Tools:** (Google, Stripe, Slack, JIRA, internal APIs...) all have different authentication, schemas, and endpoints.
- **The Result:** Developers must write custom, brittle "glue code" for every single model-tool combination. This is not scalable.

MCP (Model Context Protocol)

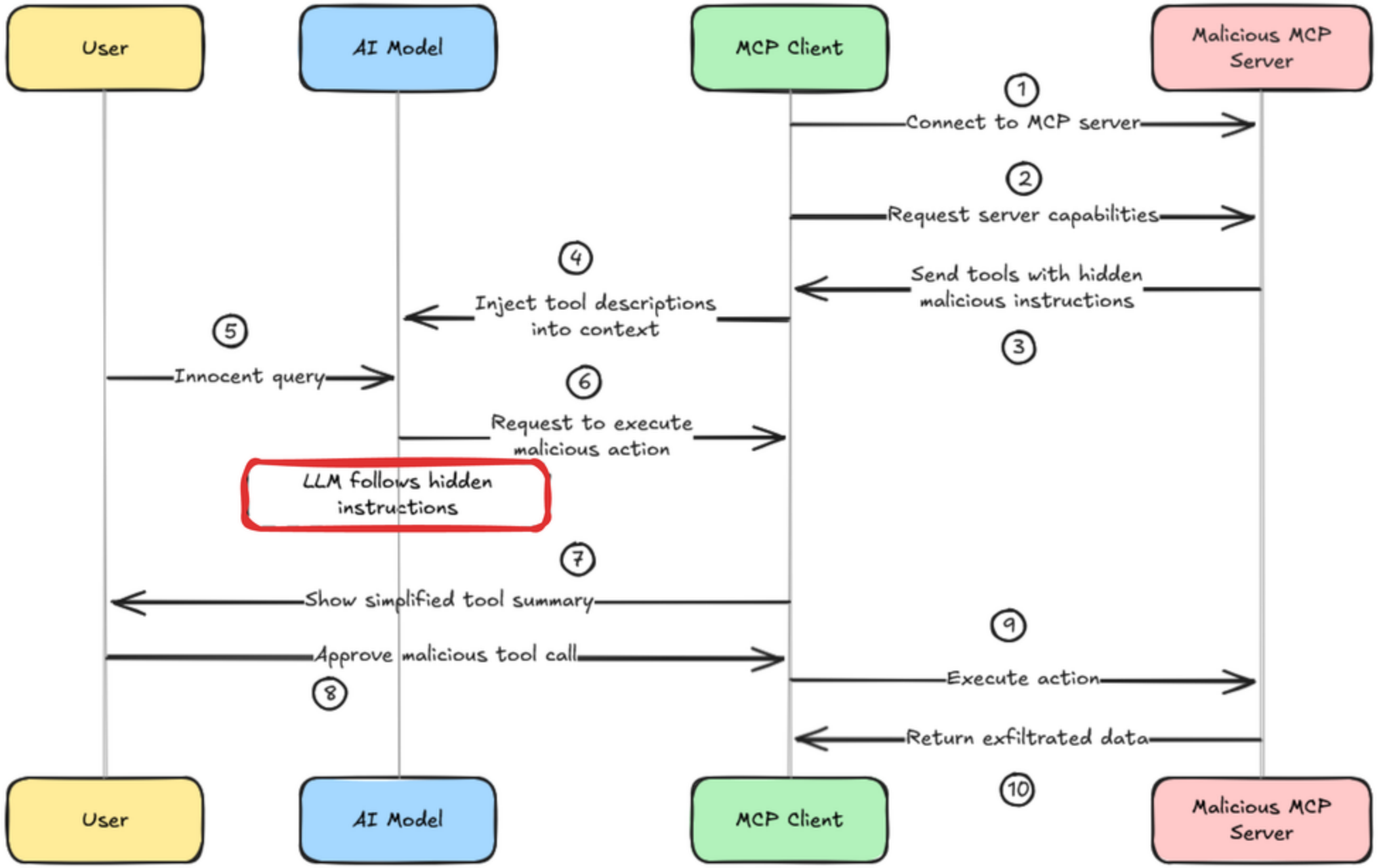
- Connecting (N) LLMs to (M) external tools/resources used to be a NxM problem
- MCP standardizes the LLM-tool communication into a N->1->M process
- Build with a client-server model
  - MCP client: the agent that needs to call tool/data
  - MCP server: a service to expose external tools and data sources





source: https://www.dailydoseofds.com/model-context-protocol-crash-course-part-8/

MCP Security Vulnerability



source: https://www.dailydoseofds.com/model-context-protocol-crash-course-part-8/

MCP: the good and the bad

- The good:
  - Unified tool protocol
  - Language and framework agnostic
  - Unifies function calling and data

- Not so good:
  - Add complexity to programmers
  - Security vulnerabilities
  - Performance overhead

Challenge 2: The Performance Problem

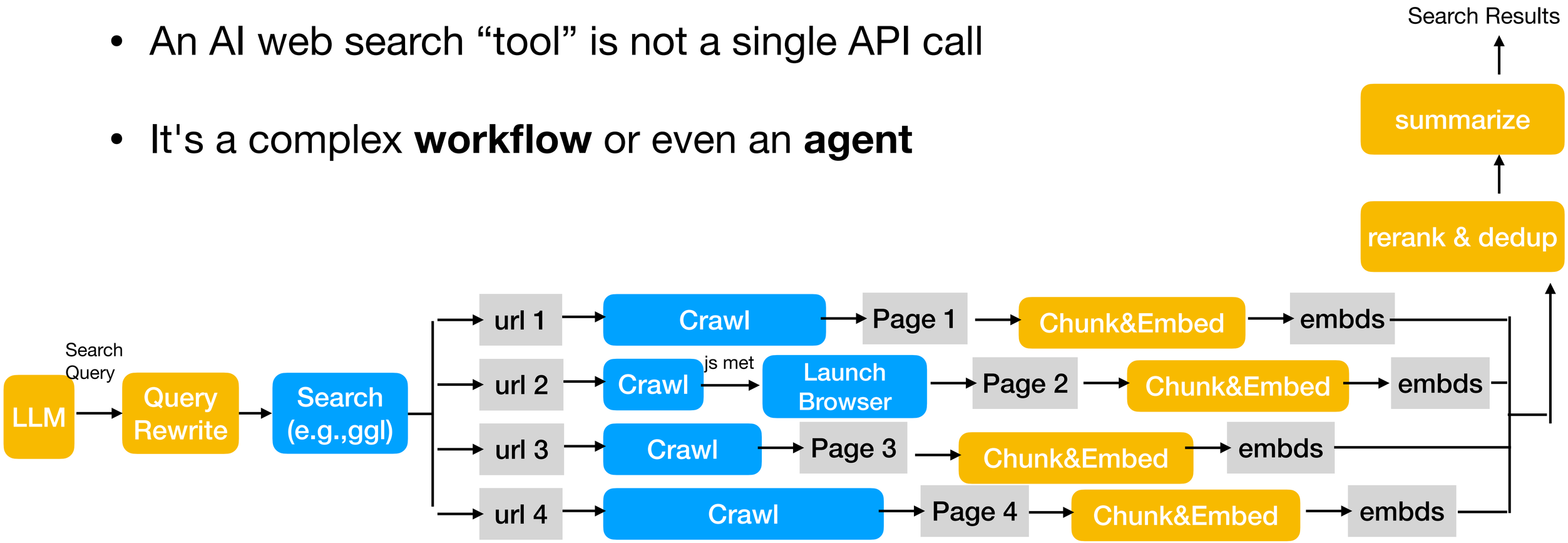
- The Problem: Sequential tool calls are slow!
- Query: ”What's the weather in San Diego, and what's the top news story on arXiv today?”
- Sequential Agent:
    1. call(weather_api, location='SD') -> (waits 1 second)
    2. call(arxiv_api, query='top') -> (waits 1.5 seconds)
    3. LLM synthesizes response -> (waits 0.5 seconds)
- Total Latency: 3 seconds

Parallel Tool Call

- When tool calls are *independent*, they can be executed in parallel.
- The LLM's reasoning step must be sophisticated.
- Instead of emitting one tool call, it emits a *list* of calls.
- The system layer executes all API calls *concurrently*.
  - `[call(weather_api), call(arxiv_api)]` -> (waits 1.5 seconds)
  - LLM synthesizes response -> (waits 0.5 seconds)
- **Total Latency: 2 seconds**

Deep Dive: Web Search Tool

- An AI web search “tool” is not a single API call
- It's a complex **workflow** or even an **agent**



What goes into all the latency?

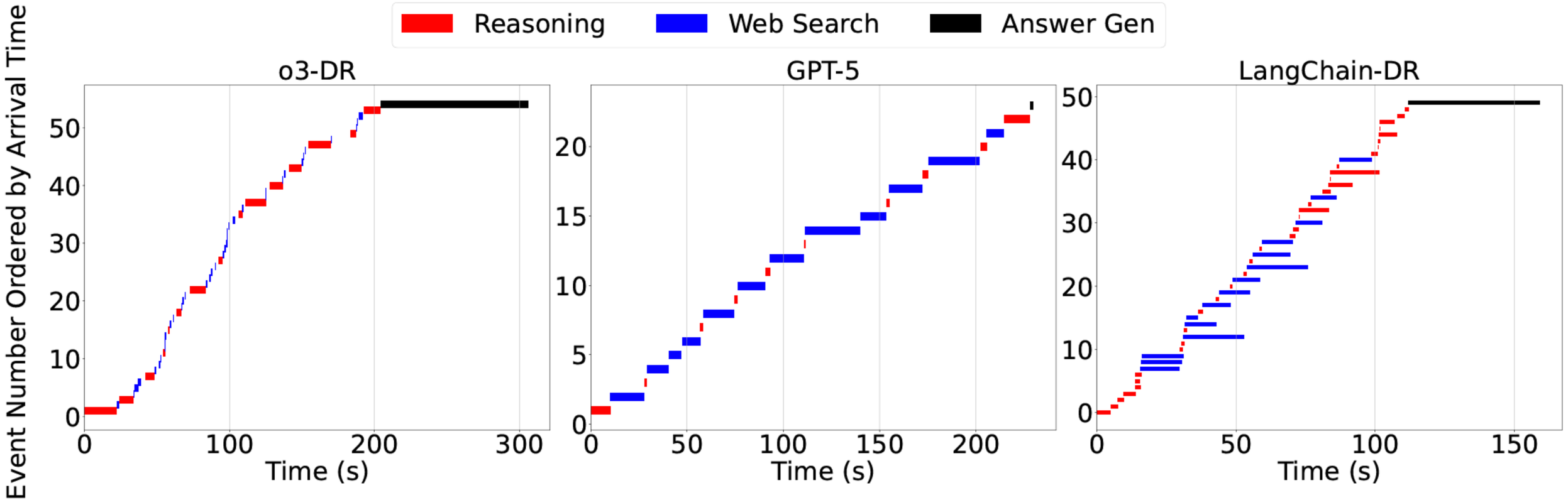


Figure 2: Timeline Comparison

Source: Qi etal. “Demystifying Delays in Reasoning: A Pilot Temporal and Token Analysis of Reasoning Systems”

BREAK TIME

Four Different Kinds of State

- Conversation history: messages and tool-call records
- Application state: files, database records, and completed actions
- Agent memory: selected facts, summaries, or past experiences
- KV cache: model activations computed from a particular token prefix

A Larger Context Window Has Costs

- More prompt tokens increase prefill work and KV-cache memory
- Long inputs can make relevant evidence harder to use
- Persistent task state still needs storage outside an individual model call
- The system chooses what to load, retain, summarize, and retrieve

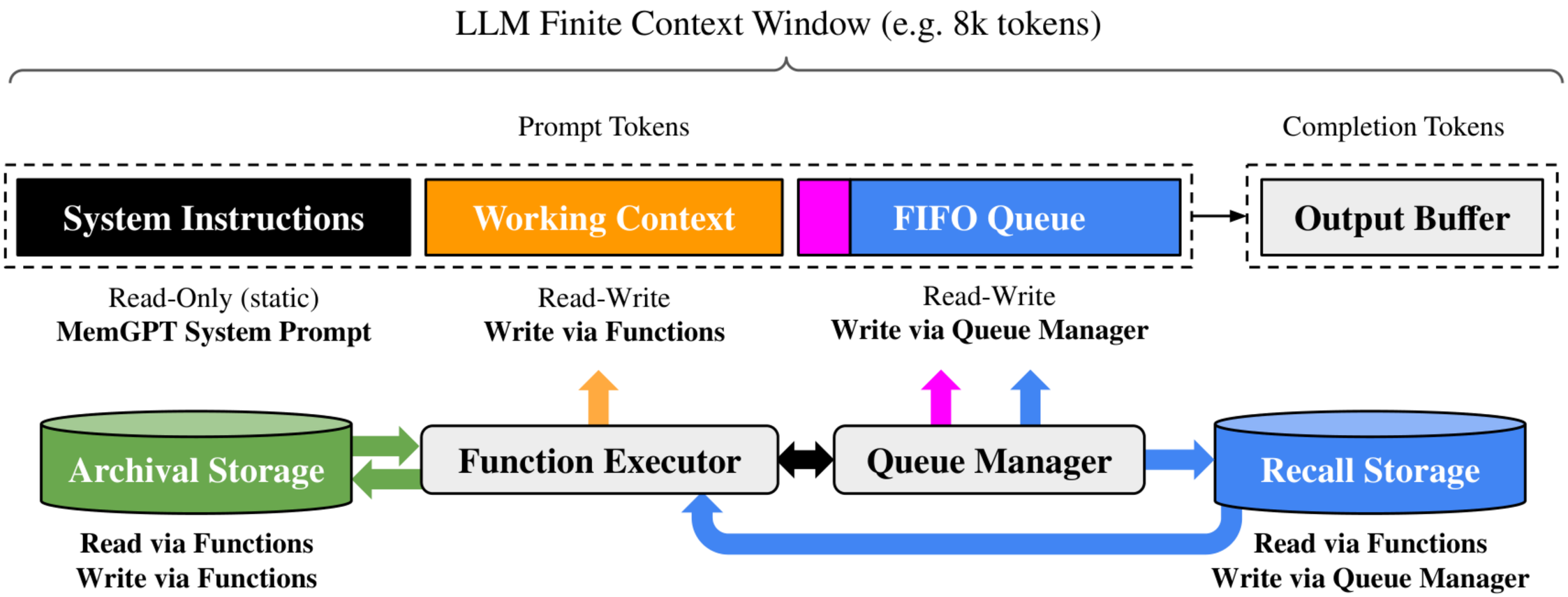
A Spectrum of Context Policies

- Windowing: keep the most recent messages
- Compaction: replace older history with a summary
- Retrieval: select records from an external store
- Agent-managed memory: the model requests reads and writes through tools

Windowing and Summarization

- A sliding window bounds the prompt but can discard needed evidence
- A summary preserves selected information using fewer tokens
- Summarization adds model work and can lose exact details
- Retain source records so the agent can recover omitted information

MemGPT: The Memory and Control Paths



Source: https://arxiv.org/pdf/2310.08560v2 (Figure 3)

MemGPT’s Memory Hierarchy

- Main context: instructions, a writable working block, and recent messages
- Recall storage: the interaction history managed by the runtime
- Archival storage: external text and documents accessed through functions
- Only information brought into the prompt is visible to the model

MemGPT’s Execution Loop

- An event appends a message and triggers model inference
- The model can edit working memory or search external storage
- The runtime executes the function and appends its result
- A requested heartbeat continues execution; otherwise the agent yields

When the Context Fills Up

- The queue manager warns the model before the context is full
- The model can preserve important facts in working or archival memory
- On overflow, the runtime evicts messages and updates a recursive summary
- Evicted messages remain in recall storage and can be retrieved later

Where the Virtual-Memory Analogy Breaks

- Retrieval selects evidence rather than reproducing an exact memory page
- Summaries are lossy; the system cannot guarantee transparent recall
- A “memory access” can require database work and another model call
- Memory policy therefore changes answer quality, latency, and cost

Persistent Instructions in Coding Harnesses

- Codex loads AGENTS.md guidance from the user and project directories
- More specific directory instructions refine the project-wide rules
- Claude Code uses CLAUDE.md and scoped rules to carry project guidance
- These files enter the model’s context again in later sessions

https://developers.openai.com/codex/guides/agents-md
https://code.claude.com/docs/en/memory

Compaction Within a Long Session

- The harness must fit instructions, messages, and tool results in finite context
- Claude Code clears older tool outputs, then summarizes when needed
- Codex supports compaction to continue work beyond one context window
- OpenAI’s Responses API can return an opaque compacted state item
- Compaction preserves working state imperfectly

https://code.claude.com/docs/en/how-claude-code-works
https://developers.openai.com/api/docs/guides/compaction

More Context or More Retrieval?

- A larger prompt spends more work up front on prefill
- Selective retrieval reduces input size but can add sequential model calls
- Repeated context edits can reduce reuse of cached prefixes
- Compare task quality and total latency at the same budget

Agent Memory Is Not the KV Cache

- A remembered fact is application data that can be edited or retrieved
- A KV cache stores activations for exact model inputs
- Reusing a cache requires compatible model state and the same prefix
- Moving an agent may move durable records without moving any KV cache

Memory Writes Need Trust and Consistency

- Tool output and retrieved text can contain attacker-controlled content
- Record provenance and separate observations from trusted instructions
- Version memory updates so concurrent tasks do not silently overwrite them
- Corrections must reach summaries and derived memories as well as raw records

What Could Go Wrong? Memory and Context

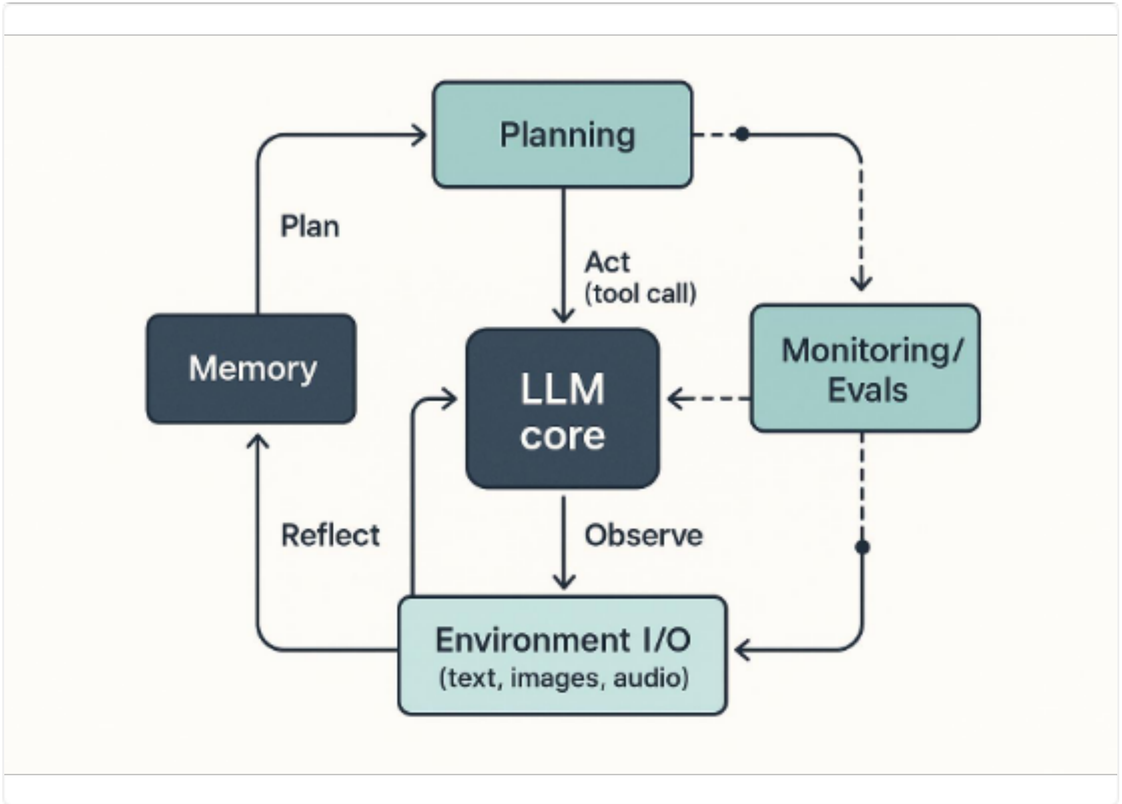
- History and summaries: could compaction drop a constraint or change its meaning?
- MemGPT: can untrusted content become a trusted, persistent memory?
- Harness memory: can a poisoned note influence future sessions?
- RLMs: what limits generated code, sub-model calls, and total resource use?
- State and caches: could stale or shared state leak another user’s data?

AI Workflow and Agent Autotuning

What is an AI Agent Workflow?

Modern agents are not single-call systems. They are **pipelines** or **graphs** of components:

- **Planning:** Decomposing complex tasks into smaller, manageable steps.
- **Memory/Context:** short-term scratchpads, long-term storage.
- **Tool Use:** Calling external APIs, running code, or searching databases.
- **Reflection:** Evaluating results, self-correction.
- **Summarization:** Summarize content and generation.



What Needs to be Tuned?

The entire system is a high-dimensional search space. We can group the "knobs" we can turn into three main levels.

Level 1: The Prompts ("Weights")

System Instructions

The "persona" or high-level directive. (e.g., "You are a master coder," "You are a helpful assistant").

Few-Shot Examples

Providing 2, 5, or 10 high-quality examples of I/O behavior to guide the model's response format.

Reasoning Style

The structure of the prompt. (e.g., "Think step-by-step," "Use the ReAct framework," "Reflect on your answer").

Level 2: The Components ("Steps")



Model Choice

The core LLM for each step. (e.g., GPT-4o, Llama 3 70B, Haiku). A massive cost vs. quality trade-off.



RAG Configuration

The parameters of your retrieval system. (e.g., Chunk size, embedding model, top-k retrieval).



Hyperparameters

The "classic" LLM settings. (e.g., Temperature, top_p, max_tokens, stop sequences).

Level 3: The Architecture ("Graph")



Workflow Structure

The literal graph of execution. (e.g., "Planner → Tools" vs. "Tool → Planner → Tool").



Chaining vs. Parallel

Do we call tools sequentially, or can we run multiple tool calls in parallel to reduce latency?



Agent Collaboration

The interaction model. (e.g., A "boss" agent delegating to "worker" agents vs. a "round-table" discussion).

Challenge: Large Search Space but Limited Budget

Assuming 3 LLM steps in an agent workflow

12 configurations in total, each with 3 options

Grid search requires: (3)^12 = 531,441 runs !!!

Assume workflow

- use gpt-4o: $10/1M tokens
- each execution output 10K tokens

➡ $53K

What's the best I can get with 128 trials ?

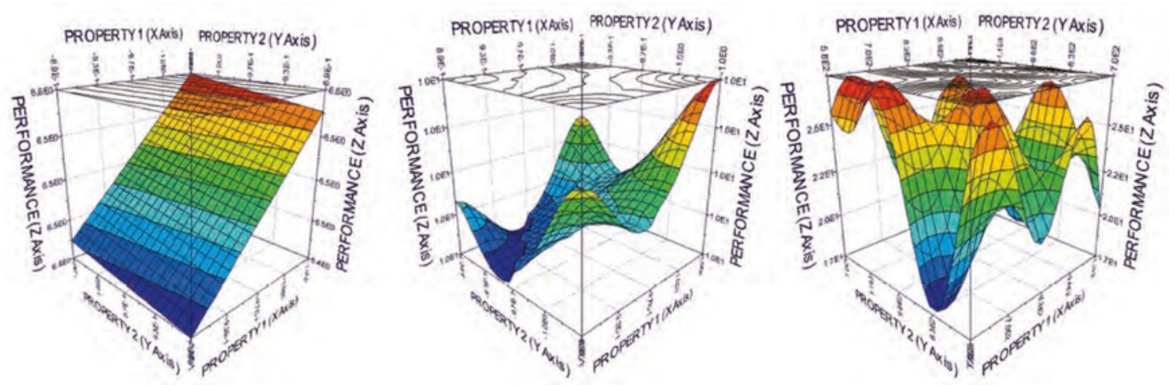
The Pain of Prompt Engineering

- Slow & ad-hoc process based on one person's intuition, not a repeatable engineering process.
- Impossible to manually optimize the vast combinatorial search space
- A "perfect" prompt for one task (or model) can fail on another.

The Multi-Objective Challenge

Usually needs to balance a set of competing goals:

- **Quality:** Accuracy, grounding, conciseness, relevance, ethics, etc.
- **Cost:** Token count, different model cost, different tool cost, etc.
- **Latency:** End-to-end latency, TTFT, tail latency, etc.



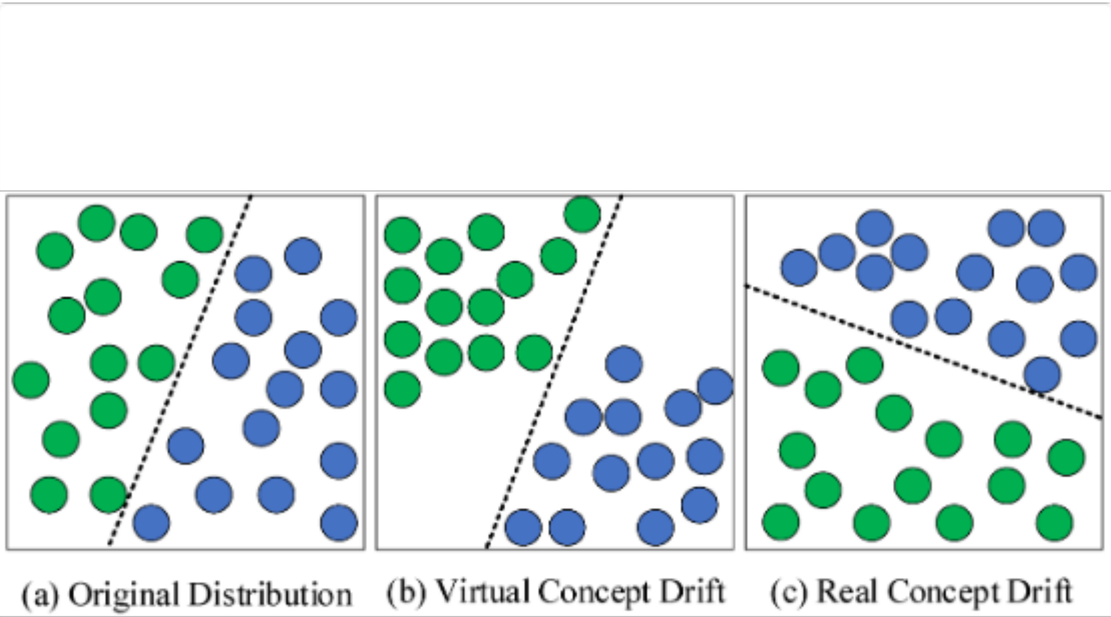
https://parametrichouse.com/optimizing-creatively-in-multi-objective-optimization/

The "Drift" Problem

Your finely-tuned, "perfect" prompt is a house of cards.

- **Model Drift:** What happens when OpenAI releases `gpt-4o-2025-05-10`? The underlying model behavior changes, and your prompts break.
- **Data Drift:** What happens when your users start asking new types of questions? Your RAG system, trained on old data, fails.

Manual tuning cannot keep up. We need a system that can **continuously re-optimize.**



From Manual to Autotuning

- Manual tuning (prompt engineering, trying different models, different architecture) is ad-hoc, does not scale, and unreliable

=> More like an “art”

- We need a more “scientific” way to scale and be reliable

Cognify: Multi-Facet AI Agent Optimization

Under submission, open source at https://github.com/GenseeAI/cognify

- AI Engineers spend a lot of effort tuning AI agents and workflows
- Cognify: aims to autotune AI agents and workflows with a small budget
- Key challenge: the search space is huge and proper searching needs $$
  - A simple 4-step workflow could need $168K and weeks to search brute force
- AdaSeek: adaptive hierarchical BO-based search
  - Uses $5 and 24 minutes to autotune the above workflow with 2.8x higher quality



"Cognify: Supercharging Gen-AI Workflows With Hierarchical Autotuning", Zijian He*, Reyna Abhyankar*, Vikranth Srivatsa, *Yiying Zhang* (* equal contribution)

Formalizing AI Workflow Autotuning

- Paraphrase to traditional ML model architecture search (e.g., NAS), operator search, weight training, and loss functions
- Key Differences:
  - Knobs in agent/workflow autotuning is not differentiable => no SGD
  - Much smaller scale than DNN (although search space is still big)

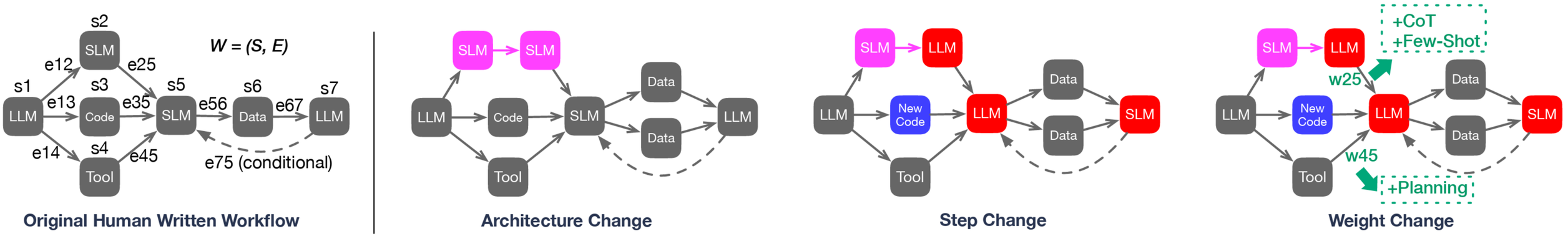


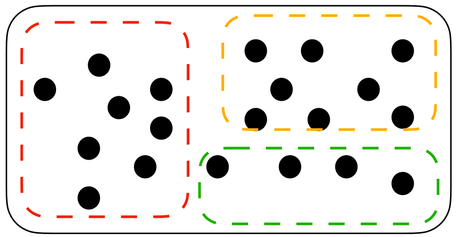
Figure 1: Gen-AI Workflows Tuning Methods. SLM and LLM represent different language models (e.g., small and large). Code, Tool, and Data represent code blocks, tool calls, and data retrieval. Dash curved lines represent loops or control flow changes. p* represents prompt optimizations or additional information added for the downstream step.

Our Approach: Adaptive Hierarchical Search

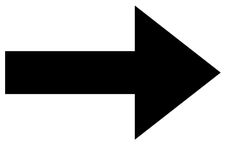
1. Hierarchical search with layered cog search space
2. Result-driven budget distribution
    A. Search space partition
    B. Search budget partition
3. Dynamic resource re-distribution

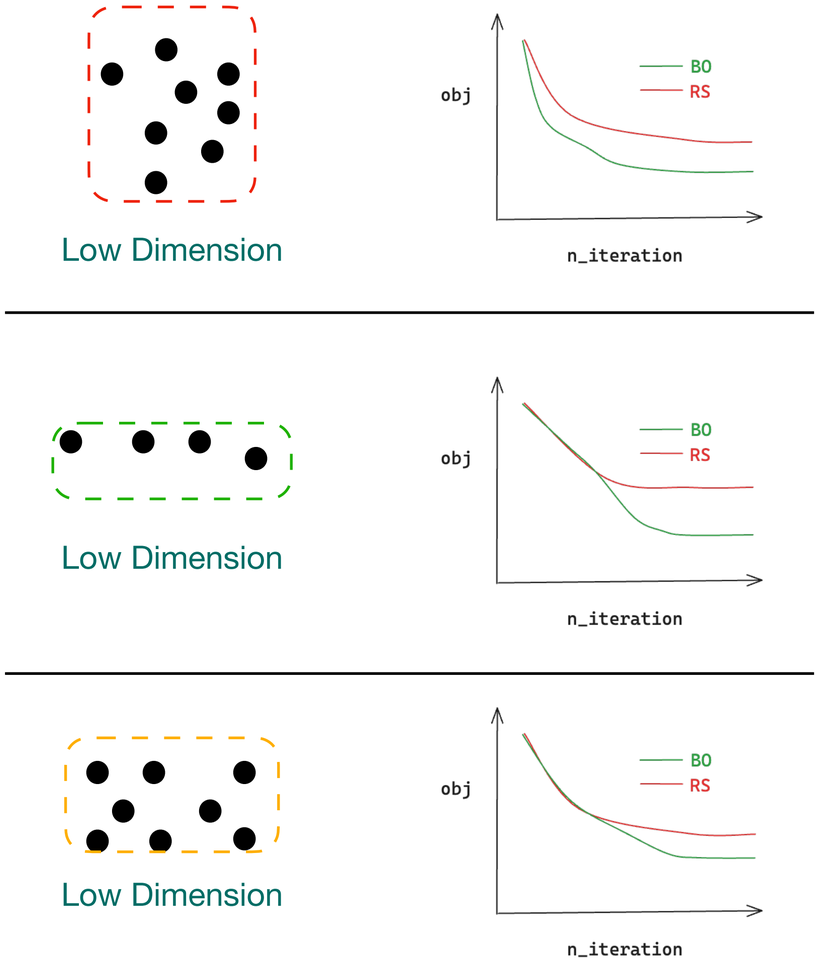
Cognify's Solution: Hierarchical Search

Flattened Search Space



High Dimension





Cognify: Organize tuning knobs (cogs) hierarchically

Top layer: Architecture Cogs

Task Decomposition

Task Ensemble

Middle layer: Step Cogs

Model Selection

Code Rewriting

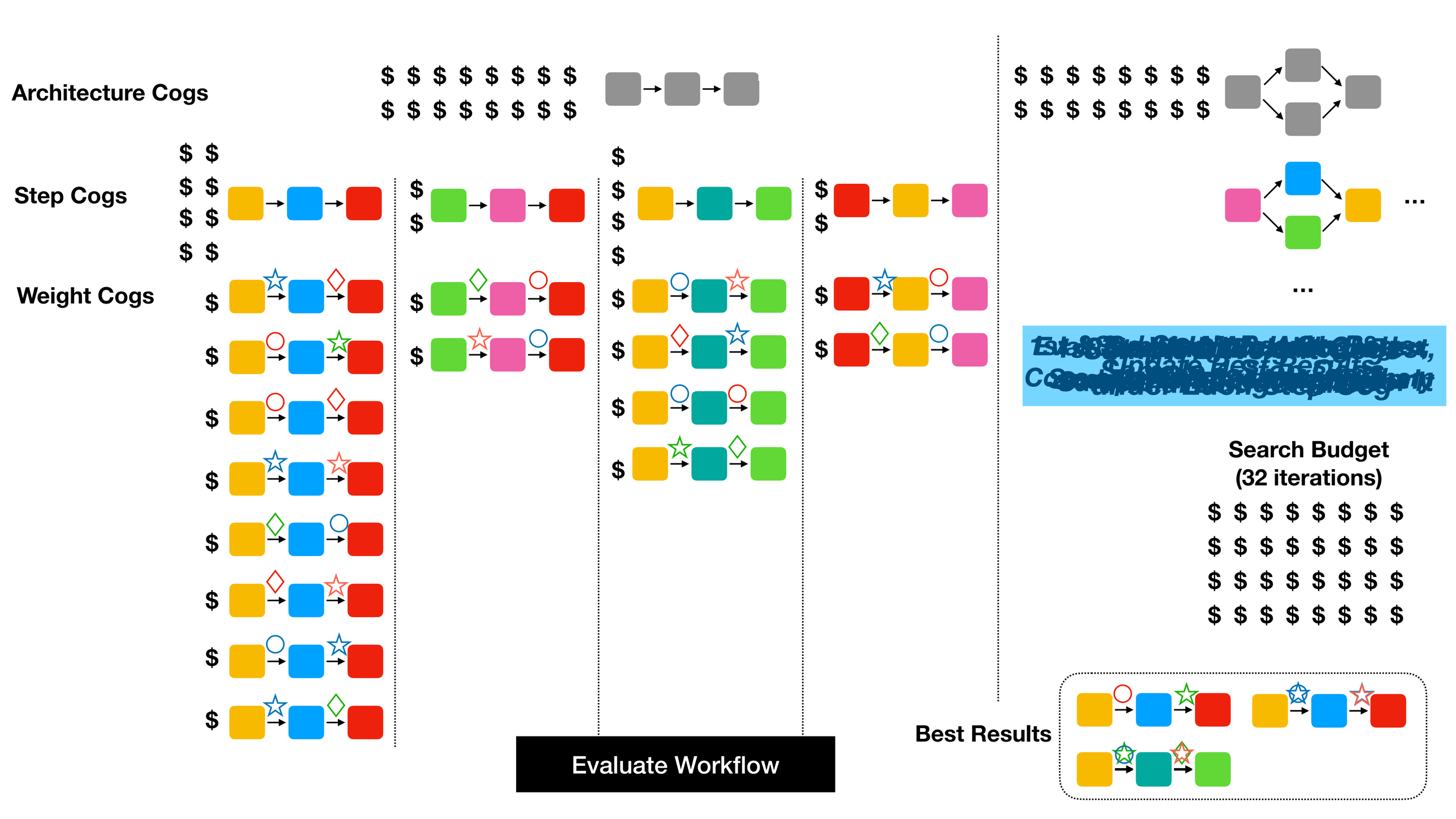
Bottom layer: “Weight” (Prompt) Cogs

Few-Shot Example

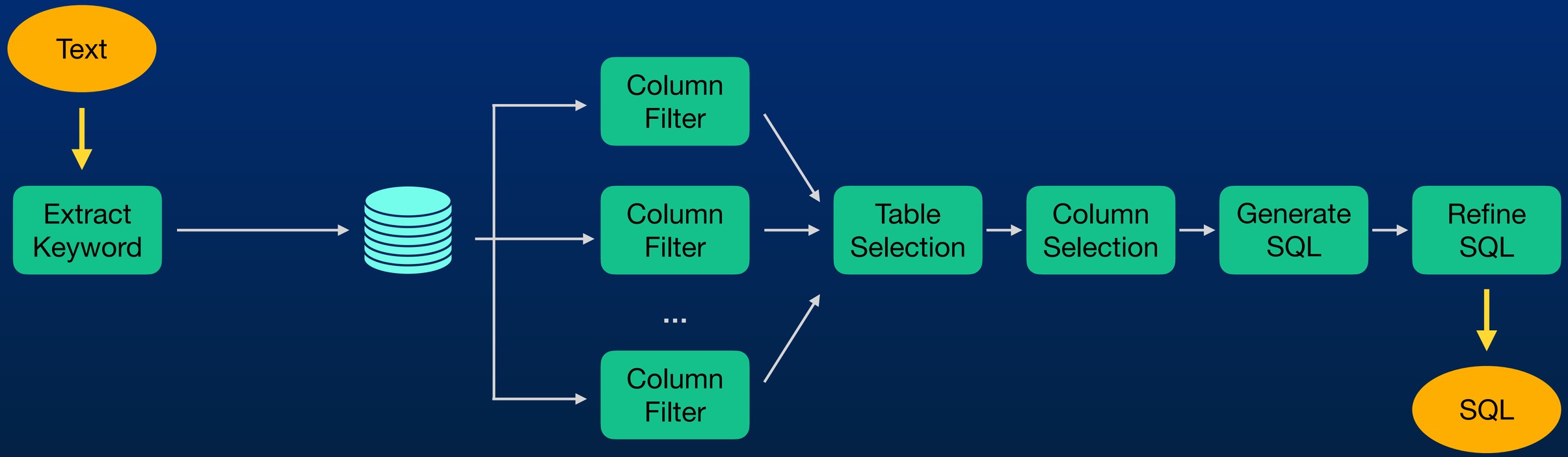
Reasoning Prompts

Cognify’s Solution: Adaptive Search

- Partition search budget across hierarchies according to layer complexity
- Direct search budget to more promising configurations using SH

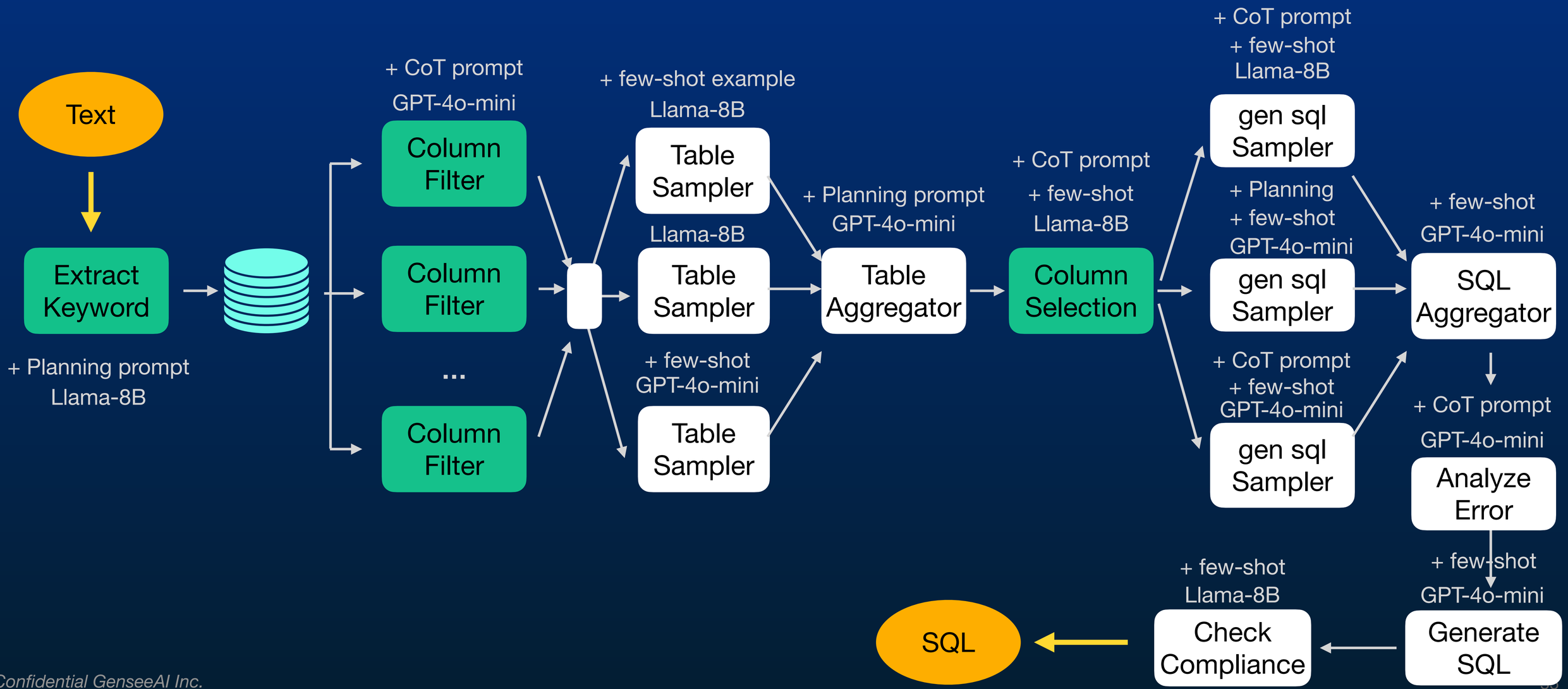

Architecture Cogs
Step Cogs
Weight Cogs
...
...
Search Budget
(32 iterations)
Evaluate Workflow
Best Results

A real gen-AI workflow: text-to-SQL

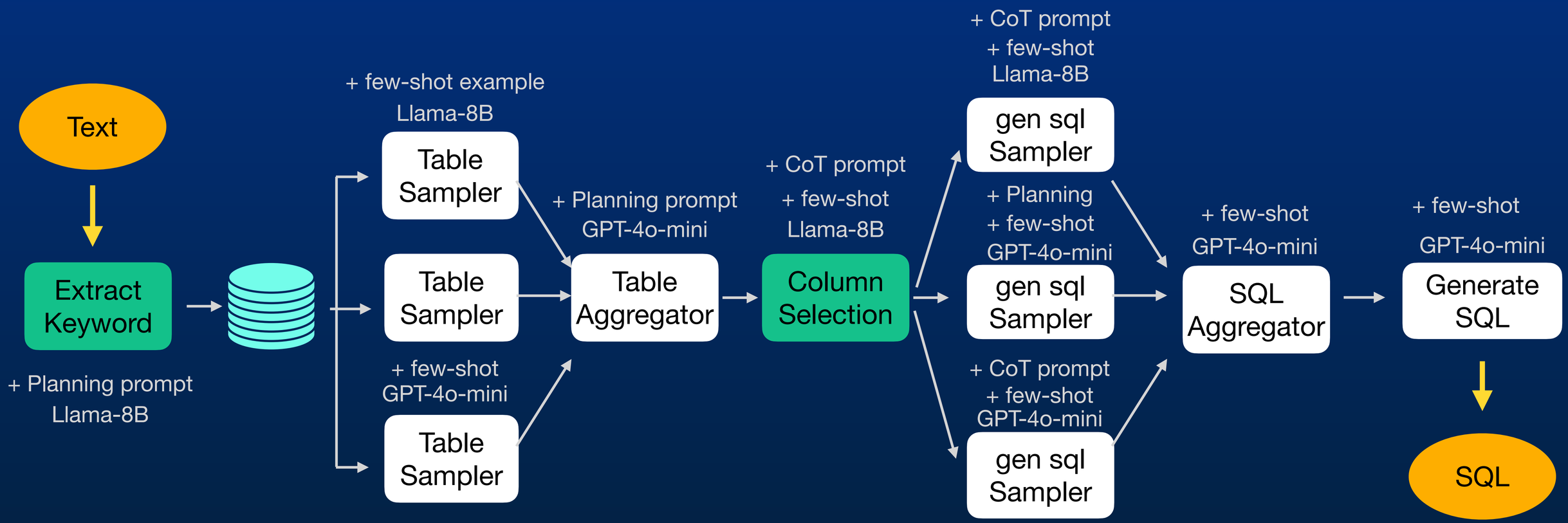


A single LLM call

Cognify's optimized text-to-SQL (training+AutoML)



Another Dignify's optimized text-to-SQL (latency-oriented)



Cognify Results

Six workloads: RAG-based QA, text-to-SQL, data visualization, financial analysis, code generation, BigBench

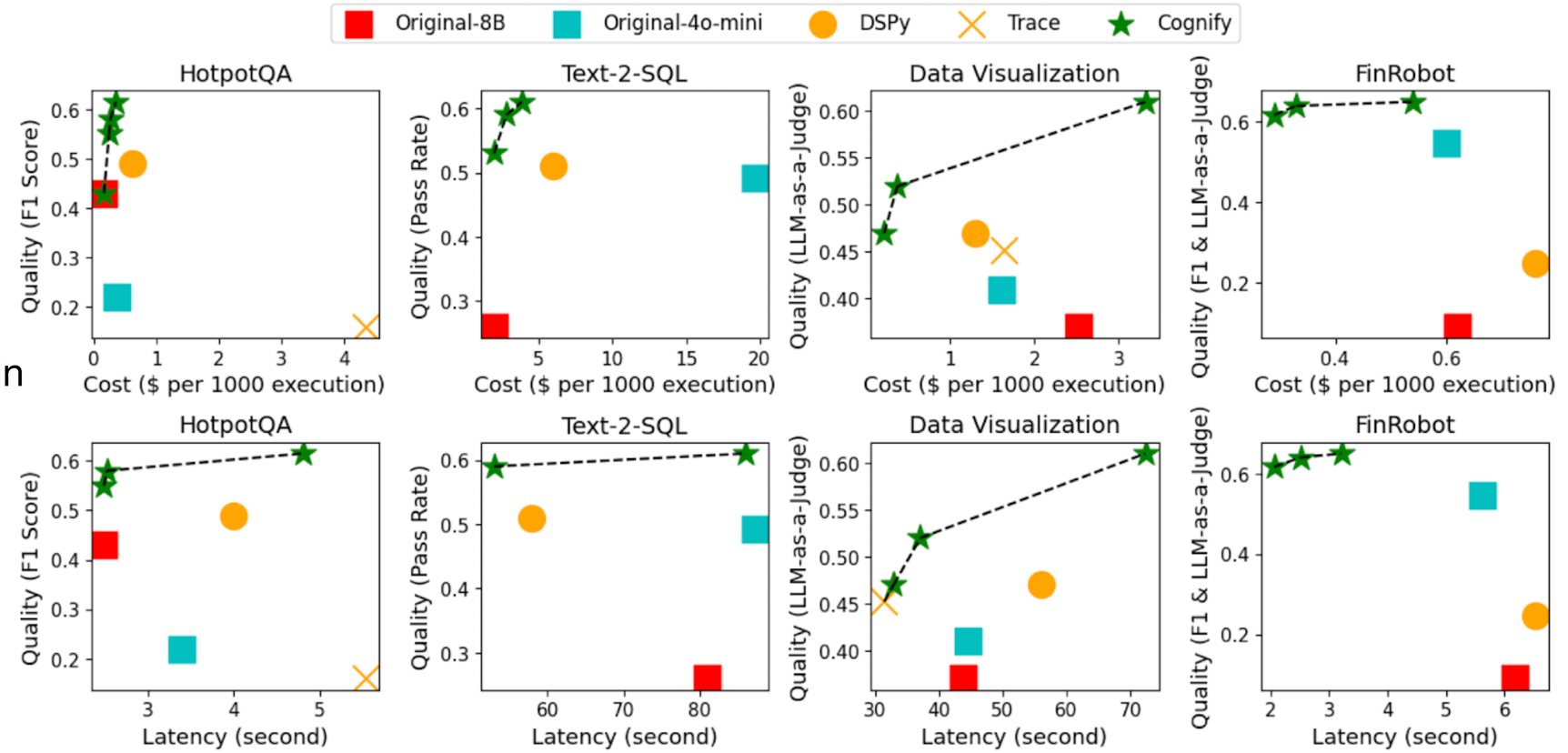
Up to 2.8x quality improvement

Up to 2.7x latency reduction

Up to 10x cost saving

Multiple choices on Pareto frontier

Low optimization costs: $1.5-$10, 8-42min (roughly half of DSPy)



Cognify Takeaways

- Formalized the problem of gen-AI workflow autotuning.
- Proposed an adaptive, hierarchical BO-based optimization algorithm, AdaSeek and demonstrated its robustness
- Significantly pushed the Pareto frontier on generation quality, cost, and latency for workflows across a wide range of domains.

Other Agent/Workflow Autotuning Methods

- DSPy (Declarative Self-improving Language Programs)
  - A workflow framework + autotuning compiler
  - Compiler invokes a “teleprompter”, an optimizer that takes the workflow program, a training set, and a metric; returns a new optimized program
    - Can swap optimizer algorithm
    - Mainly works for few-shot example selection
- GEPA (General Pareto)
  - Native support for multi-objective
  - Genetic Algorithm: Creates a set of agent configs, test them on a task, and let LLM reflect on exec traces to propose mutations to the agent.

Project Connection: Instrument One Agent Run

- Record model tokens, prefill/decode time, and tool durations
- Log context size, memory reads/writes, retries, and task success
- For edge experiments, also record energy and sustained resource use
- Keep replayable tool outputs to isolate model and runtime effects

Project Connection: Compare Policies

- Baseline: sequential tools and a recent-message window
- Change one policy: bounded parallelism, summaries, or selective retrieval
- Use the same tasks and record quality, latency, tokens, and energy
- For security studies, test malicious tool output and persistent memory contamination

Takeaways and Next Lecture

- Tools connect model decisions to executable operations
- Memory management decides which evidence enters each model call
- Runtime policy determines correctness, latency, and resource use
- Next: OS and systems-level support for agents, with TClone