SMART AI AGENTS

Don't Let Tokens Eat Up Your Budget

Designing AI agents that learn, reuse, validate, and get cheaper as they gain experience

JUAN CABRERA

LEANPUB EARLY ACCESS • PUBLICATION MASTER DRAFT 1.0 • v0.3 • SEPTEMBER 2026

Smart AI Agents

Don't Let Tokens Eat Up Your Budget

Juan Cabrera

v0.3 Publication Edition 1.0 — Early Access

September 2026

This early-access edition contains Chapters 1–3 and Appendix A. The book will expand through subsequent releases. Technical examples are educational reference implementations; production use requires security, reliability, governance, and domain-specific review.

Contents

DIGITAL EDITION Chapter and appendix titles are clickable in Word and compatible PDF readers.

About This Early-Access Edition

v0.3 is the first publication edition of Smart AI Agents. It establishes the economic problem, defines the reference architecture, and then tests the core thesis with a measured implementation.

Chapter 1 explains why repeated inference becomes an architectural and financial issue at scale. Chapter 2 defines the Smart Agent Architecture: reason when necessary, preserve what is learned, execute known work deterministically when trust is sufficient, and return to reasoning when the world changes. Chapter 3 builds and benchmarks that lifecycle. Appendix A translates the architecture into lifetime economics.

The remaining chapters will extend the same method into retrieval and memory, workflow orchestration, APIs, computer use, documents, databases, event-driven systems, executable memory, observability, and the cases where model reasoning remains the right economic choice.

Measured benchmark claims in this edition are intentionally scoped to the workloads that produced them. The objective of this book is not minimum LLM usage. It is minimum unnecessary LLM usage.

Part I

THE ECONOMICS AND ARCHITECTURE OF SMART AGENTS

Reason when necessary. Remember what was learned. Automate what becomes repeatable. Return to reasoning when the world changes.

CHAPTER 1

The Trillion-Token Problem

Why Smart Agent Design Is Now a Board-Level Decision

A business leader asks an AI agent a question:

> **Review our three largest competitors, identify meaningful pricing changes since last quarter, compare those changes with our current plans, and recommend whether we should respond.**

To the person asking, that is one request.

To the software executing it, it may be something very different.

The agent may first interpret the objective. It may determine which competitors matter, decide where to search, retrieve several pages, discover that one pricing page has moved, invoke another tool, compare old and new information, discard irrelevant material, reason about whether a price change is strategically significant, and finally compose a recommendation.

What appears at the surface as a single AI interaction can therefore become a sequence of model invocations, searches, tool calls, retrieved documents, intermediate decisions, and increasingly large contexts.

That is not an implementation accident.

It is part of what makes an agent useful.

Modern agent frameworks explicitly support this behavior. OpenAI's Agents SDK, for example, describes an execution loop in which a model can request tools or handoffs, receive their results, continue reasoning, and repeat the process until the run reaches a final output or another stopping condition.[1]

One business request can therefore produce multiple inference events.

That changes the economics of AI.

For years, many software teams developed an intuitive model of generative-AI cost that looked roughly like this:

Request → Model → Response

Estimate the number of requests. Estimate average prompt and response size. Apply the provider's token prices.

The calculation was imperfect, but understandable.

Agentic systems complicate it.

A business request is still a useful unit of work. It is no longer necessarily the economic unit of intelligence.

One request may initiate several model turns. A model turn may trigger search. Search results may become new context. The model may call an API. The API response may require interpretation. A subagent may be invoked. Context may grow. A failed attempt may be retried. Different stages may use different models.

The final answer shown to the customer can represent only the last visible artifact of a much larger computational path.

A more realistic economic model begins to resemble:

Requests × Agent Turns × Tokens per Turn × Model Price × Tool Usage

Even that is incomplete.

Cached and uncached input may be priced differently. Search and file-retrieval tools may have separate charges. Long-context rules may alter pricing. Intermediate reasoning may be billable. Tool results may increase the context carried into subsequent turns. Rate limits may constrain throughput. Latency accumulates.

For a prototype, much of this can be ignored.

For an enterprise operating agents across thousands or millions of executions, it cannot.

And that leads to the first message I want the CEO, CIO, CTO, VP of Engineering, and Chief Architect reading this book to understand:

> **This is not merely an AI-cost problem. It is an engineering problem your organization can act on now.**

Your teams do not have to wait for cheaper frontier models, a new generation of hardware, or a dramatic change in provider pricing.

There are practical techniques software engineers can begin applying to today's agentic systems:

- **cache repeated context;**
- **retrieve known information before regenerating it;**
- **identify workflows the system already knows how to perform;**

capture successful reasoning as reusable SQL, API adapters, browser workflows, parsers, rules, state machines, and executable code;

execute those capabilities deterministically when confidence is sufficient;

validate the result;

and return control to the LLM when reality changes.

Some of those techniques can reduce token consumption immediately. Others require architectural investment but can change the economics of high-volume agent workloads over time.

This book will show how to implement them, where they work, where they fail, and how to calculate whether the savings justify the engineering effort.

The premise is not that LLMs cost too much.

It is more precise:

> **An intelligent system should not repeatedly pay an intelligence premium for work it has already learned how to perform.**

When a task is genuinely new, ambiguous, changing, or uncertain, reasoning is valuable.

Use it.

Pay for it.

Give the model the tools and context it needs.

But if an agent has already discovered a reliable way to perform a stable operation, executed it successfully, validated the result, and encounters materially the same problem tomorrow, next week, or ten thousand times next month, should the organization necessarily pay an advanced model to rediscover the same solution every time?

Sometimes the answer will still be yes.

Often, it should be no.

The distinction barely matters when an agent runs ten times.

It can matter enormously when the same pattern runs ten million times.

At scale, a minor inefficiency becomes a multiplication problem:

Agents × Users × Requests × Turns × Tokens × Tools × Time

At that point, token consumption stops being a line item hidden inside an API account.

It becomes architecture.

And that is where the trillion-token problem begins.

1. One Request Is No Longer One Inference

The easiest way to understand the difference between a conventional LLM application and an agentic system is to examine what happens after the first model response.

In a simple interaction, an application sends context to a model and receives an answer:

Prompt → Model → Response

An agent introduces a loop.

The model may not return the answer. It may return a decision about what should happen next.

Perhaps it needs current information.

Perhaps it needs to query a database.

Perhaps it needs to call an internal API.

Perhaps it needs another specialized agent.

Perhaps it needs to inspect the result of something it already requested before deciding what to do next.

Conceptually, execution becomes:

Observe → Reason → Act → Observe → Reason Again → Act Again → Complete

OpenAI's Agents SDK provides one concrete example. Its runner can invoke a model, execute requested tools, process handoffs, incorporate relevant results into the run, and continue until final output or another stopping condition is reached.[1]

The important point is not that every framework behaves exactly this way.

They do not.

The important point is that repeated model invocation is a normal and intentional property of agentic execution.

A model call is no longer necessarily the application.

It may be one step inside the application.

The Multiplication Effect

Imagine, purely for illustration, an enterprise workload processing 100,000 agent requests per day.

If each request required exactly one model invocation, the system would generate 100,000 model invocations.

If the same workload averaged five model turns per request, it would generate 500,000.

Over thirty days:

15 million model invocations.

Five turns is not presented here as an industry average. Real workloads vary enormously.

The point is the multiplication itself.

A business metric such as "100,000 AI requests per day" tells us surprisingly little about the infrastructure underneath it.

Two systems could process the same number of business requests and have radically different cost, latency, and capacity characteristics because one resolves most work in one or two reasoning steps while another repeatedly loops through models, tools, searches, subagents, and growing context.

That makes the apparently simple decision—

Should the model reason again?

—an architectural decision, a latency decision, a capacity decision, and a financial decision at the same time.

Necessary Reasoning Is Not Waste

This distinction is critical.

If an agent is diagnosing an unfamiliar production outage, conducting open-ended research, reconciling conflicting constraints, evaluating new evidence, or recovering from an unexpected failure, repeated reasoning may be exactly what the application needs.

The objective is not to turn intelligent systems into rigid scripts merely because scripts are cheaper.

The problem begins when the system treats **every execution as novel**, even after experience has demonstrated that some executions are not.

Imagine a research agent that retrieves the same government statistic every month.

During its first run, it may need to determine which agency owns the authoritative data, locate the correct dataset, understand its schema, identify the relevant series, and establish how the result should be validated.

Using an LLM for that discovery may be entirely reasonable.

The system did not know where to look.

But once it has identified the authoritative API, successfully retrieved the data, found the correct field, and validated the result, what should happen next month?

One architecture says:

Search again. Rediscover the source. Reinterpret the schema.

Another says:

We have learned this. Execute what we know. Validate it. Escalate only if something changed.

The second architecture does not remove intelligence.

It preserves the result of intelligence.

Traditional software assumes that sufficiently stable behavior should eventually be encoded.

Early agentic systems often assume that behavior should be reasoned about repeatedly.

A mature architecture needs both.

It needs reasoning for the unknown.

It needs software for the known.

And because the boundary between those categories changes, it needs a mechanism for moving work in both directions:

Unknown → Reason → Validate → Learn

and later:

Known → Execute → Validation Fails → Reason Again

That mechanism will become the core architecture of this book.

For now, one principle matters:

A business request and an inference event are not the same thing.

If executives budget them as though they are, or architects observe them as though they are, the true operating behavior of an agent remains hidden.

And once agents begin looping, that hidden behavior can multiply quickly.

2. When Agentic Execution Multiplies Consumption

There is no universal agent-cost multiplier.

There cannot be.

An agent that calls a weather API and summarizes the result is fundamentally different from a multi-agent research system that delegates investigations, searches multiple sources, consolidates evidence, resolves contradictions, and produces a deeply reasoned report.

Different workloads deserve different amounts of intelligence.

But evidence from real systems shows that architecture can materially change total token consumption.

Anthropic offered a useful reference point when describing its production multi-agent research system. In its internal measurements, Anthropic reported that agents typically consumed approximately four times as many tokens as chat interactions, while multi-agent systems consumed approximately fifteen times as many tokens as chats.[2]

Those figures are striking.

They are also easy to misuse.

They do **not** demonstrate that every agent consumes four times the tokens of chat or that every multi-agent architecture is fifteen times more expensive.

Anthropic characterized them as internal observations from its own systems. The workload was research-oriented, the figures describe token consumption rather than direct monetary cost or business value, and the publication did not provide enough methodology to treat the ratios as universal constants.[2]

The useful conclusion is narrower and more defensible:

> **Changing the architecture can change inference consumption by an order of magnitude.**

That is an architectural fact worth taking seriously.

Cheap Models Do Not Fix Wasteful Architecture

Suppose two systems produce equally valuable outcomes.

System A requires one inference cycle.

System B requires a planner, three parallel researchers, a synthesizer, a critic, and a final editor.

System B may be worth the additional inference.

Perhaps it produces dramatically better results.

Perhaps the business value exceeds its incremental cost by a factor of one hundred.

If so, use it.

But architecture itself now determines a significant part of the operating economics.

That means model selection alone cannot solve agent cost.

Teams naturally focus on price per million tokens because it is visible and easily compared.

Model A costs less than Model B.

Provider X discounts cached input more aggressively than Provider Y.

Those differences matter.

But if the workflow itself consumes five or ten times more inference than necessary, optimizing only the unit price attacks the smaller variable.

A cheap model used wastefully can still produce an expensive system.

A premium model invoked selectively can sometimes produce the cheaper system.

The correct optimization target is therefore not merely:

token price

but:

unnecessary inference.

Agent Cost Has a Distribution

A second complication is variability.

A 2026 preprint examining eight frontier LLMs on agentic software-engineering tasks using SWE-bench Verified reported substantial differences in total token consumption, with observed spreads reaching as much as thirty-fold under the studied conditions. Input-token accumulation was also an important driver of overall consumption in the experiments.[3]

That result requires several qualifications.

Thirty-fold was an observed extreme, not a typical multiplier. The work concerned repository-level coding agents, not every class of agent. It was a preprint at the time of this writing rather than a peer-reviewed result. And the finding should not be generalized into a claim that all repeated runs of every agent display thirty-fold variation.[3]

But it exposes an operational characteristic worth understanding:

agent cost can have a tail.

Production engineers already understand this concept.

A database query may average 40 milliseconds while its 99th percentile is 800 milliseconds.

An API may normally return 20 kilobytes while pathological requests return 20 megabytes.

A job may usually complete in thirty seconds while unusual input keeps it alive for twenty minutes.

We do not ignore those tails because the average looks good.

Agent economics deserve the same treatment.

A system whose average inference cost is acceptable but whose unusual executions can explode in turns, context size, tool calls, or retries needs more than an average-cost dashboard.

It needs boundaries.

Those boundaries may include:

- maximum turns,
- token budgets,
- tool-call budgets,
- timeouts,
- model-routing policies,
- cost ceilings,
- circuit breakers,
- or escalation to another workflow or a human.

Most importantly, organizations need to observe the economics of the **entire execution**, not merely individual API calls.

They should eventually be able to answer questions such as:

> What is our median inference cost per completed business task?
>
> What is our 95th percentile?
>
> Which workflows have the longest reasoning tails?
>
> Which agents accumulate the most context?
>
> Which tools routinely trigger additional inference?
>
> How frequently do retries occur?
>
> Which models are used during fallback?
>
> How often does an agent rediscover something it already discovered yesterday?

And perhaps the most revealing question:

> **How much of our inference spending is purchasing new intelligence, and how much is purchasing old intelligence again?**

Imagine two executions that each consume 50,000 tokens.

The first investigates a genuinely new regulatory change, compares conflicting primary sources, and produces a conclusion the organization has never possessed.

The second repeats a stable procedure the system has successfully performed 5,000 times.

The token counts are identical.

The architectural value of those tokens is not.

The first 50,000 may be doing exactly what we purchased an intelligent model to do.

The second 50,000 may represent an opportunity to turn past reasoning into reusable capability.

This is why the thesis of this book is not simply:

Use fewer tokens.

That is a tactic.

The deeper principle is:

> **Use intelligence where intelligence creates new value. Preserve the result when the problem becomes repeatable.**

The evidence tells us that agent architecture can materially amplify consumption and that demanding agentic workloads can exhibit substantial variability.[2][3]

Together, those observations lead to a more important conclusion:

> **Inference consumption is an architectural outcome.**

Once that is true, agent efficiency cannot remain the concern of whoever happens to own the API key.

It belongs in architecture reviews.

It belongs in observability.

It belongs in capacity planning.

It belongs in financial modeling.

And as agents move deeper into enterprise operations, it belongs in conversations much higher in the organization.

3. Why This Is Becoming a Board-Level Issue

A few inefficient model calls inside a prototype do not constitute a business problem.

A team can spend a few hundred dollars experimenting, discover that half the calls were unnecessary, and reasonably conclude that optimization can wait.

Engineering time may cost more than the tokens it would save.

That logic changes when the experiment becomes infrastructure.

Imagine an organization with ten agents.

Then fifty.

Then hundreds.

Some assist employees. Some answer customers. Some monitor systems. Some research markets. Some review documents. Some generate software. Some execute operational workflows. Some call other agents.

They run not once, but continuously.

At that point, an architectural inefficiency is no longer multiplied merely by tokens.

It is multiplied by agents, users, departments, customers, workflows, execution frequency, and time.

The question changes from:

What does this API call cost?

to:

What does this architecture cost to operate at enterprise scale?

From Experiment to Operating Model

Deloitte's 2026 enterprise AI research offers a useful indication of the direction of travel. Among 3,235 director-to-C-suite leaders surveyed across 24 countries and six industries—all involved in organizational AI initiatives—23 percent reported that their organizations were already using agentic AI at least moderately when the underlying data were collected in August and September 2025. Respondents expected moderate-or-greater use to reach 74 percent within two years.[4]

The 74 percent figure is an expectation, not a measured future outcome.

And the survey did not represent a random sample of every enterprise in the global economy. Participating organizations already had working AI implementations or pilots.

Even with those limitations, the direction matters.

Organizations already investing in AI expect agentic systems to become substantially more important to their operations.

A second Deloitte study, conducted between April and June 2026, examined 501 U.S.-based leaders directly involved in agentic AI strategy or implementation at organizations already piloting the technology. Forty-two percent described their organizations as testing a small number of agents or operating a few deployments. Another 43 percent reported expanding agents across functions. Fifteen percent reported scaled, orchestrated multi-agent adoption.[5]

Those percentages should not be presented as overall U.S. enterprise adoption.

The study deliberately examined organizations already participating in agentic AI.

But that makes it useful for another reason:

it provides a glimpse of what happens **after organizations decide agents are worth deploying**.

The progression begins to look like this:

Experiment → A Few Agents → Cross-Functional Deployment → Orchestrated Agent Systems → Enterprise Infrastructure

And each step changes the economics.

Scale Converts Small Inefficiencies Into Architectural Problems

Suppose a design choice unnecessarily adds two cents to an execution.

Two cents is nothing.

A senior engineer should not spend three weeks optimizing a workflow to save two cents if the workflow runs twenty times per month.

But suppose that same execution eventually runs one million times per day.

Two cents becomes:

$20,000 per day

or approximately:

$600,000 in a thirty-day month.

The mathematics are elementary.

The difficult question is identifying **which two cents are necessary**.

If additional inference creates greater business value than it costs, eliminating it would be foolish.

If it resolves ambiguity, improves accuracy, adapts to changing conditions, prevents a costly failure, or materially improves a decision, those tokens may be an excellent investment.

If those same two cents are being spent because an agent repeatedly rediscovers something the organization already knows, the economics are different.

Scale exposes that distinction.

Traditional enterprise software is full of costs that barely matter at small scale and dominate design decisions at large scale.

A harmless database query becomes a bottleneck when executed billions of times.

A few redundant network calls become meaningful infrastructure load.

A weak cache strategy forces an organization to purchase more compute than it needs.

Good engineering has always recognized the principle:

> **Scale converts small inefficiencies into architectural concerns.**

Agentic AI does not repeal that rule.

It amplifies it.

Reasoning Is a Metered Resource

AI adds an unusual characteristic:

reasoning itself is metered.

In conventional software, developers design an algorithm and the organization can execute it repeatedly without paying the developer again every time it runs.

Execution still has compute, storage, networking, licensing, maintenance, and operational costs.

Software is never literally free.

But the intellectual work used to devise the procedure is largely separated from its marginal execution cost.

LLM-driven systems can blur that separation.

If an application asks a model to determine the procedure on every execution, some portion of the intellectual work is effectively purchased again each time.

That may be exactly what we want when circumstances change.

It becomes questionable when they do not.

This is why AI economics cannot remain solely a procurement discussion about which provider offers the lowest token price.

Architecture determines how many paid acts of reasoning the organization requires in the first place.

Adoption Can Outrun Governance

The Deloitte global study also reported that only 21 percent of respondents described their organizations as having mature governance for autonomous agents.[4]

That figure should be interpreted carefully. It applies to the surveyed population and should not be treated as a direct comparison only with organizations already reporting moderate-or-greater agent use.

Still, the contrast deserves attention.

Organizations in this AI-active population anticipated considerably broader future agent use than the level of mature autonomous-agent governance they reported at the time of the survey.

Governance discussions typically focus on security, privacy, compliance, safety, authorization, auditability, and human oversight.

Those concerns are essential.

Economic governance belongs beside them.

Who may deploy an agent that recursively invokes other agents?

What prevents an agent from entering an unnecessarily expensive reasoning loop?

Which models may be used for which workloads?

What happens when context grows beyond economically sensible levels?

How are tools budgeted?

How are retries controlled?

What happens when a premium model is automatically selected as fallback?

How do we distinguish expensive executions that produce valuable new intelligence from expensive executions that merely repeat known work?

And who can answer a deceptively simple question from the CFO:

What exactly are we buying with all these tokens?

If the answer is merely “AI,” the organization does not yet understand its own architecture.

The Board-Level Question

Calling agent efficiency a board-level issue does not mean boards should debate prompt lengths or choose embedding models.

They should not.

Boards do not configure database indexes either.

But boards care about the economic consequences of architecture when those consequences become material.

They care about operating leverage.

Gross margin.

Scalability.

Vendor concentration.

Unpredictable variable cost.

Operational risk.

And whether a technology investment becomes more efficient or more expensive as adoption grows.

The board-level question is therefore not:

How many tokens did we use?

It is:

> **Does our AI architecture become economically better as it learns, or does every increase in adoption simply multiply our dependence on paid reasoning?**

An intelligent organization should accumulate capability.

Its agents should not merely accumulate invoices.

4. The Hidden Bill Inside an Agent Run

A customer asks an agent a question.

The agent returns an answer.

The interaction looks simple enough that it is tempting to imagine the economics are equally simple:

Input Tokens + **Output Tokens** = **Cost**

That model is increasingly inadequate.

Production systems can incur charges across multiple dimensions:

- uncached input,
- cached input,
- generated output,
- intermediate reasoning,
- search,
- retrieved information,

- tool invocations,
- context storage,
- long-context pricing,
- and other provider-specific mechanisms.

Not every provider charges for every component in the same way.

That is precisely the point.

There is no longer one useful number called **the token price**.

There is a cost structure.

A Token Is Not Always Just a Token

OpenAI's API pricing documentation, for example, distinguishes input, cached input, and output pricing, while certain tools such as web and file search can introduce additional charges under their own pricing rules.[6]

A useful agent cost ledger may therefore need to distinguish among components such as:

Cost component	What it may represent
Input tokens	Context supplied to the model
Cached input	Eligible previously processed context
Output tokens	Model-generated content
Intermediate/reasoning tokens	Provider- and model-dependent reasoning work
Tool invocations	Search, retrieval, execution, and other metered services
Retrieved content	Information that becomes additional context
Cache storage	Persisted provider-side context where applicable
Long-context pricing	Different rates triggered by context size on some models

This is not a universal billing schema.

Providers, models, APIs, tiers, and enterprise agreements differ.

Prices will change.

The important lesson is structural.

If an organization records only:

Agent X processed 10,000 requests

it may know almost nothing about what Agent X actually consumed.

Even total token count may be insufficient.

Which tokens?

At what rate?

Were they cached?

How many searches occurred?

Did a request cross a long-context pricing boundary?

How many tool calls?

How much retrieved content entered subsequent turns?

How many intermediate model invocations occurred before the visible answer?

The customer sees an outcome.

Finance receives the bill for the execution path.

Search Has Its Own Economics

Web research provides a useful example.

Anthropic's web-search documentation describes a per-search charge in addition to applicable token costs, with retrieved search content capable of contributing to input usage across search iterations and later turns.[7]

The exact price is not the enduring lesson.

The structure is.

An execution may look like:

Model Decides to Search → Search Executes → Results Become Context → Model Reasons → Model Searches Again → More Context → More Reasoning

The search has a cost.

The returned content can contribute to another cost.

The reasoning over that content contributes to another.

And the next model turn may carry part of the accumulated context forward.

In an autonomous system, the model itself may participate in deciding how many times that cycle occurs.

This does not make agentic search inherently dangerous.

It makes it something that must be **observable and budgetable**.

The Customer Sees the Final Answer. The Infrastructure Pays for the Path.

Google's pricing documentation makes the distinction explicit for several managed-agent products. Its published pricing describes model inference charges that can include input, output, intermediate-input, and reasoning tokens generated during agent loops, while certain agent tools carry separate charges.[8]

Think about the implication.

A user may receive a 500-word answer.

Five hundred words are what the user sees.

They are not necessarily what the system bought.

Behind those words may be:

Planning → Search → Retrieval → Reasoning → Intermediate Context → More Reasoning → Final Generation

This gives us one of the most important economic principles for operating agents:

> **The customer sees the final answer. The infrastructure pays for the path that produced it.**

Once that principle is understood, several common optimization mistakes become obvious.

A team might obsess over shortening final responses while ignoring massive accumulated input context.

It might switch to a model with cheaper output while allowing unnecessary additional turns.

It might optimize system-prompt length while ignoring repeated searches.

It might celebrate a prompt-cache hit rate while an agent continues reasoning from scratch about the same procedure.

Those optimizations may all be useful.

None necessarily addresses the architecture.

Context Can Change the Economics

Context deserves special attention because loops naturally accumulate it.

An agent acts.

The result becomes part of its working information.

The agent acts again.

Another result arrives.

Conversation history grows.

Search results accumulate.

Tool responses accumulate.

The next model request may therefore be larger than the previous one.

Later reasoning can inherit the cost consequences of earlier reasoning.

Provider pricing can amplify that effect.

For example, xAI's documented pricing for Grok 4.6 at the time researched for this chapter included a long-context threshold at which different rates applied to the request, as well as separate charges for Web Search and X Search.[9]

The specific numbers will become outdated.

The architectural lesson will not:

> **Context size can affect unit economics.**

Production agents should therefore treat context as a managed resource, not as an infinitely growing transcript.

Cost Without Attribution Tells Us Little

The user sees:

One Request → One Result

The system may experience:

Multiple Prompts → Growing Context → Reasoning → Searches → Retrieved Content → Tool Calls → Retries → More Reasoning → Final Generation

Not every agent performs all of those steps.

But enough modern architectures can perform some combination of them that our accounting model must accommodate the possibility.

A provider invoice can tell you what you spent.

It may not tell you **why you spent it**.

A serious agent platform should eventually attribute inference and tool consumption to the application, agent, workflow, customer, task, model, tool, execution, and ideally the business outcome.

Without attribution, optimization becomes guesswork.

Imagine discovering that an agent costs $100,000 per month.

Is that expensive?

There is no way to know.

If it creates $5 million in measurable value, it may be extraordinarily efficient.

If $60,000 of that spending comes from repeatedly rediscovering a procedure that could execute safely through deterministic software, the conclusion changes.

Cost without context tells us little.

What matters is cost relative to necessary intelligence and business value.

5. Modern Agents Are Designed to Loop

At this point, it would be easy to reach the wrong conclusion.

Agents invoke models repeatedly.

Those invocations consume tokens.

Tools add cost.

Context grows.

Enterprise adoption multiplies all of it.

Therefore:

Eliminate the loops.

That would be a mistake.

The loop is not a defect in agent architecture.

The loop is one of the reasons agents are useful.

A traditional program follows logic developers have already encoded.

An agent can observe what happened, interpret the result, and change what it does next.

That ability is valuable precisely when the next step cannot be completely predetermined.

Consider a software-development agent diagnosing a failing application.

It inspects an error.

The error suggests a database problem.

It queries the database.

The database appears healthy.

That evidence changes the problem.

The agent examines logs.

The logs suggest an expired credential.

The credential is valid.

It eventually discovers that the application is connecting to the wrong environment.

No developer necessarily encoded this exact diagnostic path:

Error → Database → Logs → Credential → Configuration → Environment

The path emerged from evidence.

That is reasoning.

Removing the loop would remove much of the intelligence.

The Loop Is a Feature

Modern frameworks make this design explicit.

LangChain documents agents as models that use tools in loops until tasks are completed, with surrounding constructs for state, messages, runtime context, and persistence.[10]

OpenAI supports programmatic tool calling, tool search, deferred tool definitions, and remote Model Context Protocol integrations, allowing models to interact dynamically with external capabilities rather than function only as text generators.[11]

Microsoft's Semantic Kernel provides another example of agent invocation and higher-level orchestration, although some of the orchestration functionality documented during the research for this chapter was identified as experimental and under active development.[12]

The implementations differ.

The architectural direction is clear:

Model + **Tools** + **State** + **Control Flow** + **Feedback**

Feedback creates loops.

The agent acts.

The world responds.

The agent observes the response.

Then it determines whether its previous assumptions still hold.

That is a feature.

When Reasoning Is Worth the Tokens

Suppose an agent is researching whether a newly proposed regulation affects the business.

The information is new.

Sources conflict.

Legal language is ambiguous.

Some secondary sources misunderstand the proposal.

The agent must locate primary material, compare interpretations, identify relevant provisions, and recognize where uncertainty remains.

That is exactly the kind of problem for which we should be willing to pay an intelligence premium.

The same is true when an operations agent handles an unfamiliar outage, a cybersecurity agent investigates a novel anomaly, a coding agent encounters a repository it has never seen, or a planning agent operates under constraints that changed five minutes ago.

These are not necessarily examples of wasteful inference.

They are situations in which **inference is the product**.

The system is purchasing the ability to evaluate something that cannot safely be reduced to a predetermined procedure.

This gives us a critical boundary:

Novel. Uncertain. Ambiguous. Changed. → Reason.

That boundary should remain even in an aggressively optimized architecture.

But What Happens the Second Time?

Now change the scenario.

An operations agent successfully diagnoses an unfamiliar incident.

It discovers that a particular error indicates an expired certificate used by an internal service.

It identifies the authoritative system for checking certificate status.

It determines the approved API call.

It discovers the renewal procedure.

It executes the procedure.

The service recovers.

The result is validated.

The LLM performed valuable work.

Tomorrow, the identical condition occurs again.

One architecture says:

Start reasoning again.

Investigate possible causes.

Review logs.

Rediscover the certificate issue.

Rediscover the approved procedure.

Execute it.

The alternative says:

We have seen this before.

Retrieve the learned capability.

Verify that its preconditions still apply.

Execute the known procedure.

Validate the result.

If anything fails or no longer matches expectations, return control to the LLM.

The first architecture treats every execution as though the organization has amnesia.

The second treats successful reasoning as something that can create institutional capability.

That distinction is the heart of this book.

Reasoning Should Produce More Than Answers

Most LLM applications treat the output of reasoning as ephemeral.

The model thinks.

The application receives an answer.

The answer is consumed.

The next request begins.

What if successful reasoning could instead produce an artifact?

For a web task, that artifact might be a **Playwright workflow**.

For a database task, **validated SQL**.

For an API task, **an integration adapter**.

For document processing, **a parser and extraction schema**.

For a business process, **a state machine**.

For orchestration, **a workflow definition**.

For recurring research, **a known-source registry and retrieval procedure**.

Now the result of intelligence is not merely:

> **Here is the answer.**

It can become:

> **Here is a reusable capability for solving this class of problem again.**

That changes the role of the LLM.

The model stops being the permanent executor of every step.

It becomes one of the mechanisms through which the system discovers capabilities it did not previously possess.

Once discovered and validated, those capabilities can move into cheaper, faster, testable software.

The Important Boundary

Our division of labor becomes:

UNKNOWN / NOVEL / UNCERTAIN / CHANGED

→ Reason with intelligence

KNOWN / VALIDATED / REPEATABLE

→ Reuse capability

But the word *known* hides a hard engineering problem.

A workflow succeeded once.

Does that make it safe forever?

Of course not.

Websites change.

APIs are retired.

Schemas evolve.

Authentication changes.

Business rules change.

Sources become stale.

Customers present exceptions.

A deterministic script can return technically valid but semantically wrong output.

Smart-agent architecture therefore cannot be:

LLM Creates Script → Never Use LLM Again

That is not learning.

That is automated technical debt.

The reusable capability must remain surrounded by validation.

The system needs to determine:

- Is this truly the same task?
- Are the original assumptions still valid?
- Is the capability sufficiently fresh?
- Did execution behave as expected?
- Does the output pass validation?
- Has confidence degraded?
- Should the capability expire?
- Should it be repaired?
- Should we abandon reuse and reason again?

The architecture is therefore a cycle:

> Reason → Learn → Execute → Validate → Reuse → Detect Change → Reason Again

The model remains part of the system.

Its role simply becomes more valuable.

The Design Principle

This leads to the principle that will govern every architecture in the chapters ahead:

> **Reason when necessary. Remember what was learned. Automate what becomes repeatable. Return to reasoning when the world changes.**

Or stated economically:

> **The objective is not minimum LLM usage. The objective is minimum unnecessary LLM usage.**

Once we accept that distinction, we can ask a more precise question:

What should the system remember?

One obvious answer is the prompt.

That brings us to caching.

Caching matters.

But caching and learning are not the same thing.

6. Caching Helps—But It Is Not the Answer

When engineers discover that an application repeatedly sends the same information to an LLM, one of the first optimizations is obvious:

Cache it.

That instinct is correct.

Caching is one of the oldest and most effective ideas in computing.

Modern model providers increasingly support it. OpenAI distinguishes cached input from ordinary input for supported models and configurations. xAI documents cached-token pricing for eligible requests. Google publishes context-caching economics for applicable Gemini configurations.[6][9][13]

If a system repeatedly sends the same large context, avoiding full-price reprocessing can materially improve economics.

Use caching.

But understand what problem it solves.

Caching Answers a Different Question

Imagine an agent whose prompt contains a 30,000-token operating manual.

Every time it runs, the agent:

Reads a status code.

Determines which section of the manual applies.

Selects the corresponding remediation procedure.

Calls an internal API.

Verifies the result.

Suppose it has performed essentially the same procedure 50,000 times.

Caching asks:

> **How can we make the 30,000-token context cheaper to process again?**

Smart-agent architecture asks:

> **Why are we asking the model to rediscover the procedure again?**

Those are not the same optimization.

One reduces the cost of repeated inference.

The other asks whether repeated inference remains necessary.

Prompt Cache Versus Executable Memory

The distinction is important enough to name.

A prompt cache says:

> **We have seen these tokens before.**

Executable Memory says:

> **We already know how to perform this task.**

Suppose our government-statistics agent successfully identifies an authoritative API.

A conventional application might retain conversation history or cached context.

A learning architecture might preserve something closer to this:

```
Capability:
    Retrieve Government Statistic X

Authoritative Source:
    Agency API

Endpoint:
    /data/...

Parameters:
    date = YYYY-MM
    series = ...

Extraction:
    response.data[].value

Validation:
    expected numeric range
    expected reporting period
    authoritative source available

Freshness:
    revalidate periodically

Failure Behavior:
    return to research agent
```

It might also preserve the tested integration code.

The next execution does not need to ask:

Where do I find the statistic?

It can ask:

Is my known capability still valid?

Then:

execute,

validate,

return.

No LLM required.

Unless something changes.

Memory of Facts Versus Memory of Skills

Agent memory is frequently discussed as though remembering facts were the central challenge.

Facts matter.

A customer prefers email.

A project uses PostgreSQL.

A contract expires in November.

A previous conversation established a requirement.

Retrieval systems can bring such information back into context.

But there is another category of memory:

memory of how to do something.

Human organizations understand the difference intuitively.

Knowing that payroll closes Friday is declarative knowledge.

Knowing how to run payroll is procedural knowledge.

Knowing that an API exists is declarative.

Knowing how to authenticate, paginate, transform the response, validate it, and recover from known errors is procedural.

Agents need both.

Retrieval and RAG help answer:

> **What do we know?**

Executable Memory helps answer:

> **What do we already know how to do?**

That distinction will matter throughout the rest of the book.

Cached Reasoning Can Still Be Repeated Reasoning

Consider a computer-use agent that logs into a vendor portal, navigates to Reports, selects yesterday's date, exports a CSV file, and places it into cloud storage.

The first execution may justify an LLM-driven browser agent.

It has never seen the interface.

It identifies controls.

Discovers navigation.

Finds the date selector.

Downloads the file.

The workflow succeeds.

Tomorrow morning, prompt caching may make the static instructions cheaper.

But the architecture may still perform:

Look at Page → Ask Model What to Click → Click → Look Again → Ask Model Again → Click → Reason Again

A stronger architecture could allow the successful first run to generate:

The first execution uses intelligence to discover.

Subsequent executions use software.

The LLM returns when reality no longer matches the learned procedure.

This is not merely caching.

It is **capability acquisition**.

Three Layers of Reuse

We can now distinguish three increasingly powerful forms of reuse.

Token Reuse

What context have I already processed?

Knowledge Reuse

What do I already know?

Capability Reuse

What do I already know how to do?

Caching makes repeated inference cheaper.

Retrieval makes previously discovered knowledge available.

Executable Memory makes previously discovered **behavior** available.

They are complementary strategies.

And they lead to one of the governing ideas of this book:

> **Use intelligence to discover what you do not know. Use software to preserve what you have learned.**

The goal is not to eliminate thinking.

It is to make thinking cumulative.

7. Inference Is Also a Capacity and Latency Resource

So far, we have treated unnecessary inference mainly as an economic problem.

That is only part of the story.

Every time an agent crosses the boundary into an LLM, it consumes more than money.

It consumes time.

Capacity.

Throughput.

Part of a rate-limit budget.

Another network interaction.

Another opportunity for timeout, throttling, malformed output, transient error, or unexpected behavior.

And if the agent is operating synchronously, someone may be waiting.

> **Inference is not merely a billing unit. It is an operational resource.**

Every Reasoning Turn Has a Clock

Consider a sequential agent path:

```
User Request
     |
     ▼
Model Turn 1
     |
     ▼
Tool Call
     |
     ▼
Model Turn 2
     |
     ▼
Search
     |
     ▼
Model Turn 3
     |
     ▼
Final Response
```

Every box has latency.

Because many steps depend on the result of the previous one, not all of that latency can be removed through parallelism.

If three reasoning rounds are necessary, use them.

But understand that:

Do we need another model turn?

is also:

Do we need another latency event?

Software teams routinely optimize a database call that adds 700 milliseconds to an API.

Yet an agent may add several seconds of repeated model inference throughout a workflow without anyone asking whether those steps remain necessary.

Some of that tolerance is appropriate.

Users will wait longer for valuable research than for a database lookup.

The optimization target is not minimum latency.

It is minimum **unnecessary** latency.

Intelligence Has Capacity Limits

LLM access is also finite.

Providers expose quotas and rate limits because model infrastructure has capacity constraints.

OpenAI, for example, provides administrative access to project-specific per-model rate-limit configurations. These represent configured limits rather than a real-time meter of remaining quota, but their existence underscores that model access is a resource organizations may need to inspect and manage programmatically.[14]

At small scale, one unnecessary model call wastes a tiny amount of money.

At enterprise scale, millions of unnecessary calls can also consume capacity another workload needs.

Imagine customer-support agents, coding agents, research agents, document processors, sales-intelligence systems, and back-office automation all sharing organizational model capacity.

A repetitive low-value workflow can create contention with a high-value customer-facing one.

This is familiar infrastructure territory.

The resource is new.

The AI Equivalent of Infrastructure Utilization

Amazon Bedrock provides another indication of the operational maturity developing around inference. CloudWatch metrics include **TimeToFirstToken** for supported streaming operations and **EstimatedTPMQuotaUsage**, which estimates token-per-minute quota consumption under documented conditions.[15]

These metrics are Bedrock-specific, and the quota metric is an estimate rather than a billing ledger or perfect representation of throttling state.

The broader signal is what matters.

We are beginning to operate inference the way we operate infrastructure.

We measure latency.

Capacity.

Quota pressure.

Failures.

Consumption.

The parallels are useful:

Traditional infrastructure	Agent infrastructure
CPU utilization	Model/inference utilization
Database connections	Model/API capacity
Request latency	Agent/model latency
Query count	Model turns
Network calls	Model and tool calls
Cache hit rate	Prompt, knowledge, and capability reuse
Queue depth	Waiting agent executions
Error rate	Model, tool, and agent failure rate
Resource saturation	Quota and rate-limit pressure

These are analogies, not technical equivalences.

A token is not a CPU cycle.

But the operational principle is the same:

> **A resource repeatedly consumed at scale must be measured, attributed, and managed.**

From Token Accounting to Intelligence Accounting

Most organizations begin by asking:

How many tokens are we using?

A more mature organization asks:

Which applications are using them?

Then:

Which workflows?

Then:

Which customers?

Then:

Which agent turns?

And eventually:

> **Why?**

That final question is the most important.

Two million tokens spent investigating something genuinely unknown and two million tokens spent rediscovering a known procedure appear identical on a token meter.

Architecturally, they are completely different.

What organizations ultimately need is not merely token accounting.

They need **intelligence accounting**.

Where did paid reasoning enter the execution?

What uncertainty justified it?

What did it produce?

Was the result reusable?

Should the next execution require the same intelligence?

If reasoning consumes money, latency, capacity, and operational risk, then successful reasoning should create something durable whenever possible.

The system should learn.

And what it learns should change how the next execution runs.

8. The Architectural Response: Learn Once, Execute Many

Imagine hiring an extraordinarily capable employee.

On Monday, you give her an unfamiliar operational task.

She researches the problem.

Finds the systems.

Discovers the procedure.

Executes it successfully.

Validates the result.

On Tuesday, you give her exactly the same task.

She starts her research again.

Finds the same systems.

Rediscovers the same procedure.

Validates the same assumptions.

On Wednesday, she does it again.

And Thursday.

And Friday.

After six months, she is still rediscovering the procedure every morning as though she has never encountered it before.

No competent manager would call that intelligence.

We would call it a failure to learn.

Yet many agent architectures behave remarkably similarly.

They reason.

They search.

They use tools.

They solve difficult problems.

But successful reasoning often produces only an answer.

The answer is delivered.

The run ends.

The next execution begins with the same expensive uncertainty.

The alternative begins with a simple principle:

> **Learn Once, Execute Many.**

Not literally once in every circumstance.

Not execute forever without verification.

The phrase describes the direction of the architecture:

> **Use expensive intelligence to discover a solution when the solution is unknown. Preserve enough of what was learned that materially similar future work can begin from capability rather than ignorance.**

The Reference Architecture

The first unknown execution looks familiar:

```
UNKNOWN TASK
     │
     ▼
LLM REASONING
     │
     ▼
EXECUTION
     │
     ▼
VALIDATE RESULT
```

But successful validation is not necessarily the end.

It creates another question:

Did we learn something reusable?

If yes:

```
VALIDATED SUCCESS
       |
       ▼
CAPTURE WHAT WAS LEARNED
       |
       ▼
CREATE REUSABLE ARTIFACT
       |
       ▼
VALIDATE ARTIFACT
       |
       ▼
PROMOTE TO CAPABILITY REGISTRY
```

Now the next materially similar request follows a different path:

```
KNOWN TASK
     |
     ▼
RETRIEVE CAPABILITY
     |
     ▼
DETERMINISTIC EXECUTION
     |
     ▼
VALIDATE
    / \
   /   \
PASS   FAIL
 |       |
 ▼       ▼
RESULT  LLM RELEARNING
```

That failure path is essential.

Without it, we have not built a smart system.

We have merely generated a script.

Artifact Promotion

What moves from the intelligent path into the deterministic path?

An **artifact**.

An artifact is a reusable representation of something the agent learned.

It might be executable code, SQL, an API adapter, a Playwright workflow, a parsing function, a transformation pipeline, a state machine, a routing rule, a structured extraction schema, a known-source registry, a calculation, or a workflow definition.

Initially, it is only a candidate.

A model generated it.

That does not make it trustworthy.

It must be tested.

Validated.

Potentially sandboxed.

Observed.

Only after satisfying the system's acceptance criteria should it become reusable production capability.

We will call that process **Artifact Promotion**.

Reasoning creates the candidate.

Evidence earns promotion.

Executable Memory

Once promoted, the artifact becomes something stronger than conversational memory.

It becomes **Executable Memory**.

Traditional memory might store:

> **The monthly report is available in the Acme vendor portal.**

Executable Memory stores something closer to:

> **Here is the validated procedure for retrieving the monthly report, including authentication requirements, navigation steps, date parameters, expected file format, validation rules, known failure conditions, version information, and recovery behavior.**

The first remembers a fact.

The second remembers a capability.

The system is no longer remembering only what it knows.

It is remembering **what it knows how to do**.

Deterministic-First Execution

Once a validated capability exists, the execution policy changes.

Instead of beginning every request with:

Ask the LLM what to do

the system begins with:

Do we already possess a validated capability for this task?

If yes:

execute it.

If no:

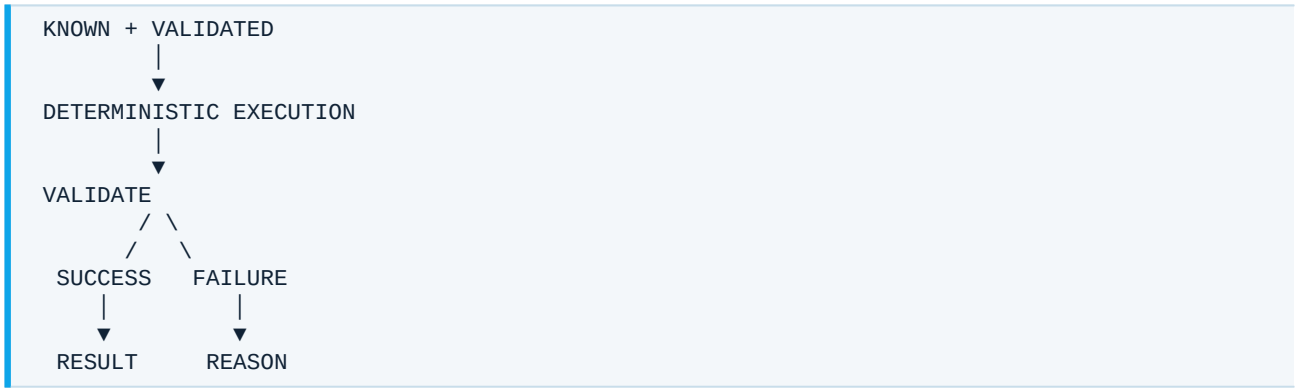
reason.

This is **Deterministic-First Execution**.

The word *first* matters.

It does not mean deterministic-only execution.

It means the architecture checks whether paid reasoning can safely be avoided before invoking it.



The LLM remains available.

It has moved from default executor toward intelligent exception handler.

LLM as Exception Handler

That phrase may sound like a demotion.

It is not.

Exception handlers deal with situations ordinary logic cannot safely resolve.

That is precisely where expensive intelligence may create its greatest value.

A browser selector disappears.

Reason.

An API schema changes.

Reason.

Semantic validation fails.

Reason.

The authoritative source moves.

Reason.

A new customer configuration appears.

Reason.

The workflow enters a state it has never seen.

Reason.

The model is no longer spending its capabilities on routine repetition.

It is being reserved for:

novelty, ambiguity, change, and failure.

Confidence-Gated Reuse

There is still a dangerous assumption hiding in this design:

How do we know that an incoming request truly matches a known capability?

Similarity is not equivalence.

Two documents may look alike while carrying different legal consequences.

Two database questions may sound similar while requiring different joins.

Two errors may share a message but have different root causes.

Capability retrieval therefore cannot safely be:

Find nearest match → Execute blindly

Reuse must be gated.

A capability may carry:

```
Capability ID
Task Signature
Supported Inputs
Preconditions
Dependencies
Version
Validation Rules
Confidence
Last Successful Execution
Failure History
Expiration Policy
Fallback Behavior
```

Before reuse, the system determines whether the capability still applies.

This is **Confidence-Gated Reuse**.

High confidence may permit deterministic execution.

Medium confidence may require stronger validation.

Low confidence may route directly to reasoning.

Risk matters.

An internal draft can tolerate a different reuse threshold than a workflow that transfers money or changes production infrastructure.

Failure-Triggered Relearning

Suppose the task was correctly identified.

The capability is retrieved.

Execution begins.

And validation fails.

A brittle system records:

FAILED

A learning system asks:

What changed?

The LLM returns—not necessarily to rediscover the entire process, but to diagnose the difference.

Perhaps one selector moved.

An endpoint changed.

Authentication added a required field.

A source schema introduced a nested object.

The system repairs the capability.

Validates the repair.

Then promotes a new version.

```
CAPABILITY v7
      │
      ▼
EXECUTE
      │
      ▼
VALIDATION FAILS
      │
      ▼
LLM DIAGNOSES CHANGE
      │
      ▼
REPAIR
      │
      ▼
VALIDATE
      │
      ▼
CAPABILITY v8
```

This is **Failure-Triggered Relearning**.

The LLM does not disappear.

It becomes the mechanism through which deterministic software adapts.

Memory With Expiration

Failure should not be the only event that returns work to intelligence.

Some capabilities should expire before they fail.

A tax rule should not be trusted indefinitely.

A pricing source should be periodically revalidated.

A website workflow may deserve scheduled testing.

An API integration should know which version it depends on.

A business policy should carry an effective date.

A statistical source should understand its expected publication cadence.

Reusable capability therefore needs freshness.

We will call this **Memory With Expiration**.

The goal is not permanent memory.

It is appropriately trusted memory.

Self-Healing Automation

Put the pieces together:

Reason → Learn → Automate → Validate → Reuse → Detect Change → Reason Again

Traditional automation is efficient but can be brittle.

Pure LLM execution is flexible but can be expensive and variable.

The architecture developed in this book attempts to combine their strengths:

LLM flexibility when reality is uncertain.

Software efficiency when reality is understood.

Validation connects the two.

That is **Self-Healing Automation**.

The Agent Efficiency Flywheel

If the architecture works, something more interesting happens over time.

Many requests are unknown at the beginning.

Unknown requests require reasoning.

Reasoning produces validated capabilities.

Validated capabilities increase reuse.

Reuse reduces unnecessary inference.

Failures reveal stale capabilities.

Relearning repairs them.

The capability library grows.

More future requests become known.

```
MORE EXECUTIONS
      |
      ▼
MORE LEARNING
      |
      ▼
MORE VALIDATED CAPABILITIES
      |
      ▼
HIGHER REUSE
      |
      ▼
LESS UNNECESSARY INFERENCE
      |
      ▼
LOWER COST + LOWER LATENCY
```

This is the **Agent Efficiency Flywheel**.

The objective is not for inference to reach zero.

It should not.

New problems appear.

Environments change.

Capabilities fail.

Some work will always require judgment.

The goal is for paid reasoning to become increasingly correlated with **actual uncertainty** rather than architectural forgetfulness.

A conventional reasoning-first system can become more expensive almost directly in proportion to how often it is used.

A learning system has the potential for different scaling behavior:

> **Usage creates knowledge. Knowledge creates capability. Capability changes the cost of future usage.**

That is the promise we will test.

This Is a Thesis, Not Yet a Proven Result

Intellectual discipline matters here.

The evidence presented earlier supports several observations:

Modern agents can involve multiple model turns.

Agentic architectures can materially increase token consumption.

Agent execution can exhibit substantial variability.

Provider billing can contain multiple inference and tool components.

Inference has latency and capacity constraints.

Those are evidence-backed observations.

The architecture presented in this section is different.

It is the **thesis of this book**.

We have not proven that Learn Once, Execute Many lowers lifetime cost for every workload.

It will not.

We have not proven that deterministic artifacts are always more reliable than repeated reasoning.

They are not.

We have not proven that the engineering cost of reusable capability always pays for itself.

It does not.

We will not ask the reader to accept those propositions on faith.

We will build the systems.

Run them.

Measure model calls.

Count tokens.

Measure latency.

Calculate deterministic execution costs.

Break their learned capabilities.

Change websites.

Change schemas.

Trigger failures.

Observe whether the agents detect those failures.

Allow the LLM to return.

Repair the capability.

Measure relearning cost.

Calculate break-even points.

And where deterministic reuse does not make technical or economic sense, we will say so.

The objective is not to prove that LLMs are expensive.

It is to determine:

> **Where does intelligence create enough value to justify its cost, and where should software take over once intelligence has done its job?**

9. What Would This Be Worth?

Architecture becomes interesting to executives when it changes economics.

So let us put numbers around the idea.

Not to manufacture a spectacular savings claim.

Not to pretend one hypothetical workload represents every agent.

And not to suggest deterministic software is free.

The purpose is to establish the financial model we will use throughout this book.

Return to the competitor-pricing agent from the opening.

It must identify price changes, normalize competitors' plans, compare them with the company's offers, and flag meaningful differences.

The first execution may deserve substantial reasoning.

The system does not know which pages are authoritative.

One competitor may expose prices in ordinary HTML.

Another may use a JavaScript application.

A third may expose an API.

The agent must distinguish promotional prices from standard prices, monthly from annual plans, and genuinely comparable tiers from misleading ones.

That is valuable discovery.

Pay for it.

The question is:

What should happen tomorrow?

Two Architectures, One Business Outcome

A conventional architecture might perform every day:

> Determine Research Strategy → Search → Navigate → Interpret → Normalize → Compare → Reason → Result

A learning architecture performs that discovery, validates what it learns, and attempts to retain the stable portions:

- **URLs**
- **API endpoints**
- **selectors**
- **extraction logic**
- **normalization rules**
- **validation criteria**
- **comparison rules**

Tomorrow's execution can begin differently:

> Retrieve Capability → Execute → Normalize → Validate → Compare

If validation fails:

Return to Intelligence.

The business receives the same category of outcome.

The infrastructure takes a different path.

That difference is what we need to price.

The Lifetime Cost Equation

Let:

```
N = number of executions
T = average reasoning turns
I = average input tokens
O = average output tokens
P = applicable model pricing
U = tool usage
R = percentage of executions eligible for reuse
F = fallback or relearning rate
C = deterministic compute, storage, and validation cost
```

Conceptually:

Traditional Agent Cost

> = Repeated Inference
> Tool Usage
> Supporting Infrastructure

while:

Smart Agent Cost

> = Initial Learning
> Artifact Creation
> Artifact Validation
> Deterministic Reuse
> Ongoing Validation
> Maintenance
> Occasional Relearning
> Supporting Infrastructure

That second equation deliberately includes costs that simplistic AI-savings calculations often omit.

- Generated software requires testing.
- Validation has cost.
- Storage has cost.
- Monitoring has cost.
- Security has cost.
- Maintenance has cost.
- Relearning has cost.

The comparison is not:

Expensive AI versus Free Software

It is:

Repeated Intelligence versus Learning Plus Reusable Execution

Follow the Pricing Agent

Assume the conventional pricing workflow consumes:

60,000 input tokens

and

6,000 output tokens

per execution.

For illustration, assume effective rates of:

$2 per million input tokens

and

$8 per million output tokens.

These are deliberately hypothetical numbers, not current provider price claims.

The model cost is:

Input:

60,000 ÷ 1,000,000 × $2 = $0.12

Output:

6,000 ÷ 1,000,000 × $8 = $0.048

Total:

$0.168 per execution

Seventeen cents.

If the agent runs once per day, the annual model cost is approximately $61.

Do not spend six months building an optimization platform for it.

Let the agent reason.

Now change the volume.

At **100,000 executions per day**:

$0.168 × 100,000 = $16,800 per day

Thirty days:

$504,000

One year:

$6,132,000

Nothing about the intelligence changed.

Only repetition changed.

Now Let the System Learn

Suppose learned capabilities eventually allow deterministic retrieval and validation at an average cost of:

$0.003 per execution

At 100,000 executions per day:

$300 per day

or:

$109,500 per year

That is still too optimistic.

Systems fail.

Suppose 5 percent of executions cannot safely complete through the learned path and require LLM involvement.

That is:

5,000 reasoning executions per day

At our illustrative $0.168:

$840 per day

Assuming the base deterministic attempt and validation still occur on those fallback executions, add the $300 daily deterministic cost:

$1,140 per day

Annualized:

approximately **$416,100**.

Now add another deliberately hypothetical:

$150,000 per year

for capability generation, platform operation, additional validation, monitoring, security, and maintenance.

The learning architecture becomes roughly:

$566,100 per year

compared with:

$6,132,000

for repeated reasoning under the assumptions above.

The illustrative difference is approximately:

$5.57 million per year.

That is not a prediction.

It is not a benchmark.

It is not a promise that your organization will save 90 percent.

It is an equation populated with assumptions so we can expose the leverage.

Change the assumptions and the conclusion changes.

If inference prices fall dramatically, the savings shrink.

If deterministic execution is expensive, the savings shrink.

If the environment changes constantly, fallback increases.

If an incorrect automated action creates unacceptable risk, additional model reasoning or human validation may be economically justified.

If the workflow runs only 500 times per year, reusable capability may make no financial sense at all.

This is not a magic formula.

It is a decision model.

Break-Even Is the Number That Matters

The most useful question is not:

How much can we save?

It is:

When does learning become cheaper than repeated reasoning?

Suppose the cost of creating, validating, securing, and operationalizing a reusable capability is:

$10,000

and deterministic execution saves:

$0.165 per run

relative to repeated reasoning.

Ignoring continuing maintenance for the moment:

$10,000 ÷ $0.165 ≈ 60,606 executions

If the workflow runs 100 times per year, do not build the capability for economic reasons.

If it runs 100,000 times per day, the break-even point arrives before the first day is complete.

Same technique.

Completely different decision.

That is why execution frequency belongs beside token count in every serious optimization analysis.

The Smart Architecture Has Its Own Tax

There is one more danger.

The capability platform itself can become expensive.

Thousands of artifacts need storage.

Versions need management.

Generated code may need sandboxing.

Dependencies must be tracked.

Old capabilities need expiration.

Validation must be maintained.

Failures need observability.

Engineers have to operate the system.

If the architecture saves $50,000 in inference while requiring $2 million in additional engineering expense, we have not created an efficiency breakthrough.

We have created an expensive hobby.

Every technical chapter in this book will therefore ask two questions:

How much inference did we avoid?

and:

What did avoidance cost us?

The relevant measure is lifetime economics.

Appendix A will bring those calculations together across multiple workload scales and, more importantly, the benchmark results generated by the systems we build in the chapters ahead.

The financial principle is straightforward:

> **The question isn't whether deterministic execution costs nothing. It doesn't. The question is whether the lifetime cost of learning plus reuse is lower than paying for repeated reasoning.**

That is what we will measure.

10. The Agent That Knows When Not to Think

Return one final time to the request that opened this chapter:

> **Review our three largest competitors, identify meaningful pricing changes since last quarter, compare those changes with our current plans, and recommend whether we should respond.**

At the beginning, it looked like one AI request.

Now we know better.

Behind the answer may be model turns, searches, tool calls, retrieved documents, growing context, intermediate reasoning, retries, latency, provider capacity, and multiple independently priced events.

The customer sees the answer.

The infrastructure pays for the path.

Perhaps every step is justified.

The competitors changed their businesses.

Their pricing models are new.

The market changed.

The recommendation requires judgment.

If so, use intelligence.

That is why we built the agent.

But suppose it runs tomorrow.

It searches for the same pages.

Rediscovers the same structures.

Determines again which plans correspond.

Reconstructs the same extraction logic.

Performs the same transformations.

Re-reasons through procedures it successfully discovered yesterday.

Then it does the same thing the day after that.

And again.

The problem is no longer that the agent lacks intelligence.

The problem is that its architecture does not allow intelligence to accumulate.

What Should an Intelligent System Become With Experience?

What should happen after an agent successfully completes a task one thousand times?

Should execution 1,001 look essentially like execution one?

Should it require the same discovery?

The same uncertainty?

The same token consumption?

The same dependence on an advanced model?

If the environment has not materially changed, that would be a strange definition of learning.

Humans do not operate that way.

Organizations do not operate that way.

Software engineering does not operate that way.

We investigate unfamiliar problems.

We create procedures.

We document what works.

We write software.

We establish APIs.

We create libraries.

We automate workflows.

We encode rules.

We test them.

And when reality changes, we revise them.

Knowledge becomes capability.

Capability becomes infrastructure.

Modern LLMs can participate in the first half of that cycle at extraordinary speed.

They can investigate.

Reason.

Write code.

Inspect systems.

Discover APIs.

Understand documents.

Plan.

Use tools.

Repair failures.

What many agent architectures lack is the second half:

> **preserving what the intelligence discovered.**

Today's Agent

Much of today's reasoning-first architecture can be simplified to:

```
REQUEST
   |
   ▼
THINK
   |
   ▼
ACT
   |
   ▼
THINK
   |
   ▼
ACT
   |
   ▼
THINK
   |
   ▼
RESULT
```

Tomorrow:

```
REQUEST
   |
   ▼
THINK AGAIN
```

These systems can be extraordinarily capable.

But they can also exhibit an unusual characteristic:

experience does not necessarily make execution cheaper.

More use simply produces more inference.

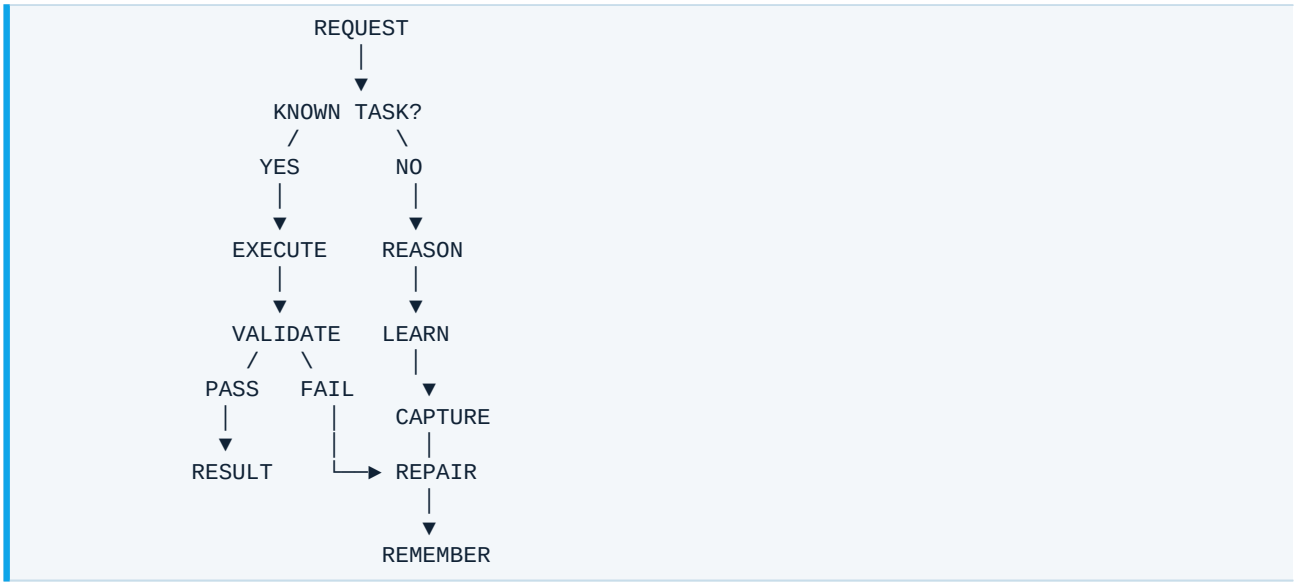
Ten times the executions can mean approximately ten times the opportunities to invoke intelligence again.

The Smart Agent

The architecture developed in this book introduces another question before reasoning begins:

> **Do I already know how to do this?**

That changes the path:



The model is no longer necessarily the first step in every execution.

It becomes one resource among several.

If capability already exists, reuse it.

If knowledge exists, retrieve it.

If repeated context can be cached, cache it.

If the task is genuinely new, reason.

If the learned capability fails, reason.

If confidence is insufficient, reason.

If the world changed, reason.

The goal is not to avoid the LLM.

It is to earn the right to avoid it after intelligence has already done its job.

Tasks Can Graduate

A useful way to think about this is that tasks can graduate.

A new task may begin as:

Reasoning-Intensive

After successful execution:

Learned

After repeated validation:

Reasoning-Assisted

When confidence becomes sufficient:

Deterministic-First

Then the environment changes:

Reasoning-Intensive Again

This is not a one-way migration from AI to code.

It is a lifecycle.

The deterministic layer provides efficiency.

The reasoning layer provides adaptability.

Validation connects them.

Intelligence Is Not Model Activity

There is a temptation to equate intelligence with activity.

More reasoning.

More agents.

More subagents.

More tool calls.

More tokens.

Sometimes more activity produces a better system.

Sometimes it simply produces more activity.

Complexity is not intelligence.

Inference is not value.

One hundred model calls that solve a genuinely difficult problem may be a bargain.

One unnecessary model call that repeats a known procedure may be waste.

The proper optimization target is therefore the business outcome:

Did the system solve the problem?

Was the result correct?

How long did it take?

What did it cost?

How much intelligence was actually necessary?

What did the system learn?

Can that learning make the next execution cheaper, faster, or more reliable?

Those are the questions that matter.

This Is Not a Book About Cheap Agents

This distinction deserves to be explicit.

This is not a book about building the cheapest possible AI systems.

Cheap systems that fail are expensive.

Fast systems that produce poor decisions are expensive.

Deterministic systems that silently execute stale procedures are dangerous.

Over-optimized systems that avoid reasoning when reasoning is necessary defeat the purpose of using AI.

The goal is not cheapness.

The goal is **economic intelligence**.

Use the expensive resource where it creates value.

Do not use it simply because the architecture forgot everything.

A smart system should be capable of saying:

> **I have never seen this. I should think.**

Then:

> **I know this. I should execute.**

And, critically:

> **I thought I knew this, but validation failed. I should think again.**

That is a more useful definition of a smart agent.

The Strategic Asset the Model Provider Does Not Own

There is another implication for enterprise leaders.

If two companies use the same frontier model, they do not necessarily possess the same AI capability.

They may purchase intelligence from the same provider.

But one architecture may forget almost everything between executions.

The other may continually convert successful reasoning into reusable organizational capability.

Over time, the second company can accumulate something the model vendor does not provide:

a proprietary library of validated ways of getting work done.

Its workflows.

Its API knowledge.

Its source mappings.

Its extraction logic.

Its validated queries.

Its decision rules.

Its recovery procedures.

Its browser automations.

Its transformations.

Its operational knowledge.

The underlying frontier model may be available to every competitor.

The accumulated capability is not.

That may become one of the most important strategic distinctions in enterprise AI.

The model provides general intelligence.

The organization's architecture determines whether that intelligence leaves something durable behind.

The Question Has Changed

We began this chapter with the natural question:

How much does the model cost?

That question is no longer sufficient.

The more important question is:

Why did this execution need the model?

If the answer is:

because the problem was new;

because evidence conflicted;

because the environment changed;

because judgment was required;

because confidence was low;

because deterministic execution failed;

then inference may have been exactly the right choice.

But if the answer is:

because that is simply what the agent does every time

then we have found an architectural opportunity.

And unlike many discussions about future AI economics, that opportunity does not require waiting for cheaper models or another technological breakthrough.

Software engineers can act on it now.

They can instrument model calls.

Measure repeated reasoning.

Identify stable workflows.

Retrieve before regenerating.

Cache when reasoning remains necessary.

Promote validated reasoning into executable artifacts.

Route known tasks toward deterministic execution.

Validate those executions.

Expire stale capability.

And return to intelligence when the system encounters uncertainty or change.

Those are the techniques we will implement throughout the rest of this book.

From Principle to Engineering

The concept sounds straightforward.

The engineering is not.

What constitutes a reusable capability?

How do we recognize that an incoming request matches one?

What belongs in Executable Memory?

When is retrieval sufficient?

When should an artifact become executable code?

Where should generated code execute?

How do we prevent unsafe Artifact Promotion?

How should confidence be calculated?

How do we validate without spending as many tokens as we saved?

How long should capability memory live?

How do we detect change before an outdated procedure causes harm?

When should the LLM regain control?

How do we repair capabilities automatically?

How do we measure the savings accurately enough that finance can trust the result?

Those are engineering questions.

And they are where this book goes next.

> **The most efficient intelligent system is not the one that thinks fastest. It is the one that remembers when it no longer needs to think.**

Chapter 2: Stop Paying Your Agents to Think Twice

In the next chapter, we turn that principle into an implementable architecture.

We will define what constitutes a reusable capability, what belongs in Executable Memory, how an agent recognizes a known task, when Deterministic-First Execution can be trusted, how validation gates reuse, when learned capabilities should expire, and how the LLM regains control when reality changes.

Then we will assemble those pieces into the reference architecture that drives the rest of this book:

Request → Capability Lookup → Deterministic Execution When Known → Validation → LLM Reasoning When Unknown or Failed → Capture → Promote → Reuse

The problem is now defined.

Chapter 2 builds the system that solves it.

Notes and References

1. OpenAI, "Running Agents," OpenAI Agents SDK documentation.
Describes the agent-run loop in which models may request tools or handoffs and the runner continues until final output or another stopping condition. Used here as a concrete implementation example rather than a claim that every agent framework follows identical control flow.

2. Anthropic, "How We Built Our Multi-Agent Research System," Anthropic Engineering.
Anthropic reported internal observations that agents typically consumed approximately four times the tokens of chat interactions and multi-agent systems approximately fifteen times the tokens of chats in the workloads discussed. These figures are workload- and implementation-specific and are not treated as universal multipliers.

3. "Tokenomics of Agentic Coding," 2026 preprint examining eight frontier LLMs on SWE-bench Verified, arXiv:2604.22750.
The study reported substantial execution-to-execution token variability, including an observed maximum spread reaching approximately thirty-fold under the studied conditions. The result is benchmark-specific and should not be generalized to all agents or workloads.

4. Deloitte, *State of AI in the Enterprise 2026*.
The underlying global survey included 3,235 director-to-C-suite leaders involved in AI initiatives across 24 countries and six industries. The chapter uses its reported current agentic-AI use, anticipated future use, and autonomous-agent governance findings with the survey-population and expectation-versus-realization qualifications stated in the text.

5. Deloitte Insights, 2026 U.S. agentic-AI research involving 501 senior-manager-to-C-suite leaders at organizations already piloting agentic AI.
Used to illustrate stages of adoption among organizations already engaged in agentic AI. The percentages are not presented as estimates of overall U.S. enterprise adoption.

6. OpenAI, API Pricing documentation.
OpenAI publishes differentiated pricing categories including input, cached input, and output tokens for supported models and documents separate pricing mechanisms for certain tools. Pricing is time-sensitive and should be verified before financial use.

7. Anthropic, Claude API Web Search Tool documentation.
Documents web-search usage charges in addition to applicable token consumption and explains

how search-result content can contribute to input usage across agent interactions. Pricing and billing mechanics are time-sensitive.

8. Google, Gemini API Pricing documentation.
Documents inference and tool-pricing structures for listed managed-agent products, including applicable treatment of input, output, intermediate-input, and reasoning tokens. The relevant rules are product-specific and subject to change.

9. xAI, API Pricing documentation for Grok 4.6 and search tools.
Documents standard and long-context pricing treatment along with separately priced search tools. Used here to demonstrate that context size and tool use can alter request economics; current rates should be verified before financial modeling.

10. LangChain, JavaScript Agents documentation.
Documents agents combining models, tools, state, messages, runtime context, and iterative execution. Used as an implementation example rather than evidence that all agent frameworks use the same architecture.

11. OpenAI, Tools and Agents documentation.
Documents programmatic tool calling, tool search, deferred tool definitions, remote MCP integrations, and related mechanisms through which models interact with external capabilities.

12. Microsoft Learn, Semantic Kernel Agent Architecture documentation.
Documents direct agent invocation and agent orchestration capabilities. Some orchestration functionality referenced during research was identified as experimental or under active development and is not treated as production-stability evidence.

13. Google, Gemini API context-caching pricing documentation.
Illustrates separate economics for eligible cached context and, in applicable configurations, cache storage. Specific rates vary by model and change over time.

14. OpenAI, Admin API Project Rate Limits documentation.
Documents an administrative endpoint for retrieving configured project-specific model rate limits. These configured limits should not be interpreted as a real-time measure of remaining quota or current consumption.

15. Amazon Web Services, Amazon Bedrock CloudWatch metrics documentation and related AWS announcement.
Documents metrics including TimeToFirstToken for supported streaming requests and EstimatedTPMQuotaUsage for applicable Bedrock operations. EstimatedTPMQuotaUsage is an operational estimate rather than a billing ledger and varies by supported model, API, and region.