How to become a Staff AI Engineer

Chapter 17: Reasoning Models and Test-Time Compute for AI Engineering

Celso Sousa
LinkedIn: https://www.linkedin.com/in/celso-sousa/?locale=en-US
Portfolio: https://celsosousa.com.br/project-portfolio/
Ebook: https://celsosousa.com.br/ebook-staff-ai-engineer/

Chapter Objective

Reasoning Models introduce a new dimension of scaling in artificial intelligence systems. In addition to investing computational capacity during pretraining and post-training, it becomes possible to allocate more compute during inference itself to increase the probability of correctly solving complex problems. This mechanism is frequently referred to as **test-time compute** or **inference-time compute**.

In conventional inference, the system typically receives an input, executes the model, and produces a response. In reasoning-oriented architectures, execution may use additional resources to decompose the problem, explore hypotheses, produce multiple solutions, consult tools, retrieve evidence, verify results, and correct failures before producing the final response.

At first, this shift may appear to be a model characteristic, but its implications are architectural. When the amount of work begins to vary per request, reasoning directly affects latency, throughput, GPU utilization, cost, reliability, observability, capacity planning, security, and unit economics.

For a Staff AI Engineer, therefore, the objective is not to maximize the amount of reasoning. The objective is to determine when additional compute actually increases the value produced by the system and how that compute should be controlled within product constraints.

Several principles should remain present throughout this chapter:

- **Reasoning can be treated as search for a solution:** Complex problems frequently have multiple plausible paths, and additional compute makes it possible to explore alternatives before the system commits to an answer;
- **Decomposition creates value only when it reduces net complexity:** Separating requirements, evidence, calculations, and decisions can make a problem more verifiable, while excessive decomposition introduces coordination, context, and new failure points;
- **Verification often provides stronger confidence than additional reflection:** When an external mechanism can test a solution, its evidence tends to be stronger than repeatedly asking the model to evaluate its own answer;
- **Test-time compute has real operational cost:** Additional reasoning increases tokens, latency, accelerator consumption, tool calls, and workload variability;

- **Compute should be allocated adaptively:** Trivial cases should not consume the same resources as difficult problems, high-value decisions, or high-risk situations;
- **Reasoning does not replace retrieval or deterministic software:** Thinking for longer does not produce missing facts, nor does it turn probabilistic generation into a superior alternative to operations that can be executed exactly;
- **Autonomy must remain constrained by the architecture:** The model may decide how to investigate a problem or suggest an action, but authorization, budgets, invariants, and critical controls must remain outside probabilistic behavior.

These principles change the engineering question. Instead of asking only whether more reasoning improves quality, we need to ask: **what quality improvement do we expect to obtain by spending more compute on this task, and does that improvement justify the increase in cost, latency, risk, and operational complexity?**

This is the perspective that should guide every system based on Reasoning Models.

17.1. Reasoning Models as a System Capability

Reasoning Models are models or systems designed to use additional processing before finalizing a response or decision. This processing does not need to take a single form and may occur either within the model or through the architecture surrounding it.

The main strategies include different ways of expanding the work performed during inference:

- **More processing before the final response:** The model uses greater computational effort to analyze the task before finalizing the output;
- **Multiple candidate solutions:** The system produces different responses or trajectories and subsequently compares their results;
- **Search over alternatives:** Different paths can be explored instead of immediately using the first trajectory found;
- **Tool use:** Calculators, code execution, search, databases, and APIs can provide facts or execute operations that should not depend on probabilistic generation;
- **Verification:** A candidate response can be submitted to tests, constraints, or other evaluation mechanisms before being accepted.

The architectural effect of these strategies is significant. In traditional serving, requests within the same category tend to have relatively similar costs. In reasoning systems, two apparently similar requests may produce completely different workloads. One may terminate after a single inference, while another may execute multiple steps, tool calls, verifications, and retries.

This makes the workload heterogeneous across multiple dimensions:

- **Latency:** Total execution time becomes dependent on execution depth;
- **Token consumption:** Different tasks may consume dramatically different token volumes;
- **Accelerator utilization:** Some requests remain small, while others occupy capacity for significantly longer periods;
- **External resource consumption:** Tools, databases, search APIs, and other services may be invoked repeatedly;
- **Cost:** Unit cost stops being approximately uniform and becomes dependent on the executed trajectory.

As a consequence, QPS alone is no longer an adequate representation of load. Capacity planning needs to consider the distribution of work performed per request and, especially, its long tails.

17.1.1. Training-Time Scaling and Test-Time Scaling

There are two broad ways of increasing the computational capacity available to a model-based system.

Training-time scaling invests resources before deployment. This may involve larger models, more data, additional training, or other mechanisms that increase the capability obtained during training.

This strategy has several important properties:

- **Additional capability becomes available to every request:** Every call uses the more powerful model regardless of the actual difficulty of the task;
- **Much of the investment occurs upfront:** The primary cost is introduced during development and training, although larger models may also increase serving cost;
- **Application architecture may remain relatively simple:** The model receives an input and produces an output without necessarily executing an adaptive workflow.

Test-time scaling, by contrast, dynamically increases resource usage during inference. This approach changes both system economics and architecture:

- **Compute can be concentrated on difficult cases:** Simple requests remain relatively inexpensive, while complex problems receive more processing;
- **Cost becomes more strongly task-dependent:** The same API may exhibit substantial differences in tokens, execution time, and consumed resources;
- **The application becomes more sophisticated:** Budgets, verification, routing, retries, termination, and escalation become part of the runtime.

These strategies should not be treated as mutually exclusive alternatives. A mature architecture combines model capability with test-time compute.

The question is no longer “Should we use a better model or increase reasoning?”. A more useful question is: **what combination of model capability and inference-time compute maximizes task success within our latency and cost budgets?**

17.1.2. When Test-Time Compute Actually Helps

Additional reasoning tends to provide greater value when a problem requires the composition of multiple decisions and when the system can transform additional processing into better evidence. Some classes of problems exhibit this behavior more frequently:

- **Multi-step problems:** Coding, debugging, mathematics, architecture, research, and planning may require multiple conclusions to be connected before reaching the final result;
- **Problems with multiple plausible alternatives:** Exploring different hypotheses reduces the risk of premature commitment to an inadequate solution;
- **Highly verifiable problems:** The stronger the available verifier, the greater the system’s ability to use additional compute to generate and test solutions;
- **High-value or high-risk problems:** Small improvements in correctness may justify substantial costs when the economic cost of a wrong decision is high.

However, many problems attributed to “insufficient reasoning” actually belong to other categories:

- **Missing knowledge is a retrieval problem:** If the required information is unavailable, increasing reasoning does not create a reliable source;
- **Exact computation is a tooling problem:** Deterministic operations should be delegated when specialized software can execute them with stronger guarantees;
- **Insufficient capability may be a model problem:** Some tasks may exceed the capabilities of the selected model, and simply increasing processing may not eliminate that limitation;
- **Known sequences are workflow problems:** When the correct steps are already defined by the business process, introducing probabilistic planning adds variability without proportional benefit.

A Staff-level diagnosis should therefore distinguish among a **knowledge problem, reasoning problem, tool problem, model-capability problem, and workflow problem**. This separation is essential. Without it, test-time compute can become an expensive way of masking problems that should be fixed elsewhere in the architecture.

17.2. Decomposition, Planning, Search, and Refinement

Reasoning capability should not be interpreted simply as “thinking for longer.” More robust systems structure additional processing through mechanisms that reduce complexity, explore alternatives, and introduce checkpoints. Among the most relevant mechanisms are decomposition, planning, multiple sampling, search, and iterative refinement.

17.2.1. Decomposition

Decomposition divides a complex problem into smaller units that can be analyzed with greater independence. This approach is particularly useful when different parts of a task have distinct requirements, information sources, or verification mechanisms.

An architecture analysis, for example, can explicitly separate different dimensions:

- **Requirements:** Determine the behavior the system needs to deliver;
- **Scale:** Defines volume, concurrency, and expected growth;
- **Data architecture:** Analyzes sources, state, consistency, and data movement;
- **Reliability:** Considers failures, retries, recovery, and availability;
- **Security:** Identifies trust boundaries, authorization, and attack surfaces;
- **Economics:** Evaluates costs, unit economics, and expected return.

This separation can produce important benefits:

- **Reduced local complexity:** Each step operates on a smaller subset of variables;
- **Greater verifiability:** Different subproblems can have their own success criteria;
- **Better tool selection:** Some steps may require retrieval, others computation, databases, or code execution;
- **Greater observability:** Failures can be attributed to specific stages rather than appearing only in the final result.

However, decomposition is not free:

- **Every new step introduces coordination:** State, evidence, and decisions must cross additional boundaries;

- **More steps increase the failure surface:** Each component may fail, produce incorrect information, or exceed its timeout;
- **More intermediate results expand context:** Previous outputs often need to be propagated to subsequent steps;
- **Excessive decomposition fragments coupled decisions:** Problems that should be evaluated jointly may lose important relationships when artificially separated.

The architectural principle should be simple: **decompose while separation reduces the system's net complexity. Stop when the added coordination becomes greater than the benefit obtained.**

17.2.2. Planning and Replanning

Planning organizes actions before or during execution. It is especially useful when dependencies exist between decisions and when the order of operations affects the outcome.

A common pattern is: **plan → execute → observe → replan.** Separating planning from execution can introduce important controls:

- **Plans can be inspected before actions are performed:** This is particularly important when tools can modify external systems;
- **Human approval can be introduced before critical operations:** Low-risk parts remain automated without removing oversight from sensitive actions;
- **Expected cost becomes more visible:** An excessively long plan can be interrupted before consuming the entire budget;
- **Structural failures can be detected early:** Incorrect assumptions may be identified before producing downstream effects.

A plan, however, should not be treated as an immutable contract. Reasoning systems frequently operate in environments where APIs become unavailable, expected data does not exist, or initial hypotheses are invalidated by new evidence. Replanning represents the ability to change strategy in response to these changes.

There is an important difference between **replanning** and **retrying**. Replanning changes the strategy because the evidence has changed. Repeatedly executing the same action while expecting a different result merely creates a more expensive loop.

The planning horizon must also be calibrated:

- **Long horizons make it possible to consider global objectives:** But they depend on a larger number of assumptions remaining valid;
- **Short horizons reduce speculative planning:** The system plans only the next actions for which it has sufficient evidence;
- **Checkpoints allow observations to be incorporated:** The environment can be reassessed before execution continues;
- **Event-driven replanning reduces unnecessary work:** The plan only needs to change when a relevant assumption is no longer valid.

In uncertain environments, short-horizon planning combined with frequent checkpoints tends to reduce dependence on fragile predictions. The role of the Staff AI Engineer is to balance global visibility with adaptability.

17.2.3. Multiple Sampling, Best-of-N, and Search

Another way to consume test-time compute is to explore more than one trajectory. In multiple sampling, the system produces several solutions to increase the probability that at least one of them is adequate.

This technique may improve quality, but it introduces costs and limitations:

- **Compute grows approximately with the number of samples:** The improvement must be large enough to justify multiplying inference;
- **Samples are not truly independent:** The same model may repeat systematic errors across multiple trajectories;
- **Insufficient diversity wastes compute:** Producing several variants of essentially the same solution adds little value.

Best-of-N adds a selection stage. N candidates are generated, and a scorer or verifier attempts to identify the best one.

Performance then depends on two components:

- **The generator needs to produce good candidates:** No selection mechanism can choose a correct solution if it never appears in the candidate set;
- **The scorer needs to recognize quality:** A weak verifier may reject the best solution and select an inferior response.

For this reason, Best-of-N must be evaluated end-to-end. Increasing the average quality of candidates does not guarantee an improvement in the response delivered to the user if the selection mechanism is unreliable.

A related variant is **self-consistency**. When multiple trajectories converge on the same conclusion, that agreement may serve as an additional signal.

However: **agreement is not proof.** Systematic errors can produce high agreement across multiple trajectories.

More complex problems may also be represented as search over reasoning paths. In this scenario, the system needs to manage the relationship between breadth and depth:

- **Breadth explores more alternatives:** It reduces the risk of remaining trapped in a poor hypothesis selected early;
- **Depth investigates one trajectory more deeply:** It becomes necessary when the solution depends on a long sequence of decisions;
- **Pruning limits combinatorial explosion:** Unpromising paths must be eliminated before consuming the entire budget;
- **Search bounds limit resource usage:** Branching factor, depth, time, and cost need externally imposed limits.

Without pruning and budgets, a search architecture can rapidly consume resources without producing proportional improvement.

17.3.3. Self-Correction and Iterative Refinement

Self-correction allows the model itself to review a previous output. A simple pattern is: **draft → critique → revise.** This strategy can improve properties such as completeness, consistency, and constraint adherence.

Some failure types respond particularly well to this mechanism:

- **Missing fields can be detected:** A review can compare the output against the expected schema or specification;
- **Explicit contradictions can be reduced:** Different parts of the response can be compared against one another;
- **Forgotten requirements can be reincorporated:** The model can revisit the specification before finalizing;
- **Presentation issues can be corrected:** Formatting or organization can be improved without repeating the entire execution.

The structural limitation is that the same model that previously failed to recognize an error may remain incapable of identifying it during review. There is also another risk: **refinement may degrade a response that was originally correct.**

Consequently, self-correction should be used proportionally:

- **Self-correction works well as a refinement mechanism:** It can improve average quality and adherence;
- **Independent verification is preferable when correctness is critical:** External evidence reduces dependence on the same failure mode;
- **Iterations must be bounded:** Repeating revisions indefinitely does not imply convergence toward a correct response.

The purpose of these techniques is not to construct the longest possible trajectory. It is to use additional structure only when it transforms compute into a higher probability of success.

17.3. Verification and Reliability

Verification is one of the most important mechanisms for transforming test-time compute into real reliability. The fundamental idea is to separate two concepts that are frequently conflated: **generating a solution** from **producing evidence that the solution satisfies the expected criteria.** This separation prevents linguistic confidence from being treated as correctness.

17.3.1. Preference for Deterministic Verification

Whenever a property can be verified by deterministic software, that strategy should be prioritized. Several examples illustrate this principle:

- **Code can be compiled and tested:** Running tests produces stronger evidence than asking the model whether its implementation is correct;
- **Structured outputs can be validated against schemas:** A syntactically plausible response should not enter another system without validation;
- **Numerical results can be recomputed:** Calculators and specialized engines reduce unnecessary uncertainty;
- **Data can be queried from the source of truth:** A database query is preferable to the model's parametric memory when the fact is transactional;
- **Invariants can be evaluated automatically:** Limits, allowed states, and critical rules should have enforcement outside the model.

A particularly important principle for AI Engineering is: **do not use probabilistic intelligence to verify a property that deterministic software can test directly.** This principle simultaneously reduces error, cost, and evaluation complexity.

17.3.2. Model-Based Verification

Not every property has an objective verifier. Text quality, coverage, coherence, relevance, or appropriateness may require semantic judgment. In these cases, another model can act as a verifier or scorer.

This approach is useful, but it must be interpreted as a probabilistic signal:

- **The verifier also has an error rate:** Its score does not represent ground truth;
- **Generator and verifier may have correlated failures:** Similar models may agree on the same incorrect conclusion;
- **Calibration remains necessary:** We need to understand how automated judgment relates to human or external evidence;
- **False positives and false negatives produce different costs:** Approving an incorrect response may be much more serious than rejecting a valid response in certain workloads.

Therefore, LLM-based verification does not eliminate the need to evaluate the verifier itself.

17.3.3. Local and End-to-End Verification

Verification also needs to operate at different levels:

- **Local verification reduces error propagation:** Tool arguments, schemas, intermediate states, and partial results can be validated immediately;
- **End-to-end verification measures task success:** A workflow may contain individually valid components and still produce an inadequate global result.

Robust architectures use both perspectives. Local verification protects internal boundaries; end-to-end evaluation verifies whether the system actually achieved its objective.

17.3.4. Failure Modes Introduced by Reasoning

Additional reasoning may increase average task success while simultaneously introducing new failure modes. One of the most important is **error propagation**. An incorrect conclusion produced early may become an assumption for every subsequent step. The longer the chain, the greater the blast radius of that failure may become.

Checkpoints and intermediate verification help interrupt this process before the incorrect hypothesis contaminates the entire execution. Another common problem is **reasoning loops**.

The system may continue repeating:

- **Similar queries:** Without obtaining new evidence;
- **Equivalent analyses:** Reformulating the same conclusion;
- **Redundant tool calls:** Consuming APIs without making progress;
- **Repeated revisions:** Changing details without improving the result.

The runtime needs progress-detection mechanisms. Number of steps, evidence novelty, and repeated actions can indicate that execution has entered a loop.

It is also possible to obtain a correct answer through fragile reasoning. The result may be correct by coincidence, shortcut, or dependence on an irrelevant characteristic of the input. Such a system has low robustness. Small changes in problem formulation may produce significant failures.

The inverse is even more dangerous: **an incorrect response accompanied by highly plausible reasoning.** Fluency, detail, and narrative coherence can increase human confidence in an incorrect conclusion. In critical systems, apparent explainability cannot replace evidence.

Another relevant failure mode is **confirmation bias**. Once the system chooses a hypothesis, new information may be interpreted primarily as confirmation of the initial decision.

Several strategies reduce this risk:

- **Generate alternative hypotheses:** Prevents excessively early commitment;
- **Search for contradictory evidence:** Forces the system to confront its conclusion;
- **Apply falsification:** Explicitly asks what evidence would demonstrate that the hypothesis is incorrect;
- **Introduce independent checkpoints:** Allows assumptions to be reassessed before execution continues.

These techniques are particularly valuable in debugging, research, incident analysis, and other contexts in which an incorrect initial hypothesis can direct the remainder of the investigation. The central conclusion is that more reasoning does not eliminate failure modes. It changes how errors emerge and propagate. The longer and more autonomous the execution, the more necessary explicit engineering of verification, termination, and recovery becomes.

17.4. Adaptive Compute, Routing, and Resource Control

One of the most important decisions when placing Reasoning Models into production is avoiding the use of the same reasoning budget for every request. Workloads have different levels of difficulty, risk, and economic value. An architecture that applies deep reasoning indiscriminately pays the maximum cost even when the task could have been solved through a simpler alternative. An efficient pattern is to start with the least expensive option and increase capability as evidence indicates that it is necessary.

A cascade can follow this logic:

1. A smaller model produces an initial attempt;
2. A verifier evaluates whether the criteria were satisfied;
3. If it fails, the system increases reasoning or uses a more capable model;
4. Extreme cases can be escalated to another tier or human review.

This behavior is a form of **adaptive compute**. When properly designed, it produces important benefits:

- **Easy cases remain inexpensive:** A large portion of traffic can be resolved without frontier capability;
- **Compute follows difficulty:** Expensive resources are concentrated in situations with greater marginal benefit;
- **Risk can influence rigor:** A technically simple task may receive additional verification when the consequence of an error is high;
- **Unit economics improve:** The organization stops paying for maximum capability on every interaction.

The difficult part is determining when to escalate. Several signals can guide this decision:

- **Verification failure:** An output that violates objective constraints requires recovery or escalation;
- **Retrieval insufficiency:** Lack of sufficient evidence may prevent a reliable response and require another path;
- **Task complexity:** Different problem categories may have different difficulty priors;

- **Decision risk:** The potential cost of an error may justify a larger budget even when the task appears simple;
- **Low system confidence:** Internal signals may contribute to routing, provided they are calibrated and not treated as ground truth.

Adaptive compute transforms the reasoning budget into a runtime policy.

17.4.1. Model Routing and Cascades

Not every workload requires the same model. A platform can operate multiple tiers:

- **Small models for simple tasks:** Reduce latency and cost when the required capability is low;
- **General-purpose models for intermediate cases:** Handle workloads requiring greater flexibility without justifying the most expensive tier;
- **Advanced reasoning models for complex cases:** Concentrate superior capability on situations in which it produces greater value;
- **Human escalation for critical exceptions:** Keeps cases beyond system limits outside unrestricted automation.

Poor routing produces two failure modes:

- **Under-routing reduces quality:** A difficult case sent to insufficient capability may fail or require multiple retries;
- **Over-routing reduces efficiency:** A trivial task sent directly to the most expensive tier consumes resources without proportional benefit.

A typical cascade may take the form: **small model → verification → larger model → human.** This architecture allows most of the volume to remain within economically efficient tiers without losing access to greater capability when necessary.

The relevant KPI is no longer simply model accuracy. A measure closer to the system objective is: **successful tasks per dollar within the latency SLO.**

17.4.2. Budgets

Unbounded reasoning is an architectural risk. Every workflow must have budgets imposed externally to the model.

Different resources need to be bounded:

- **Token budget:** Defines how much textual processing may be consumed;
- **Step budget:** Limits reasoning, planning, refinement, or search cycles;
- **Tool-call budget:** Prevents loops from consuming APIs, databases, or external resources indefinitely;
- **Time budget:** Protects latency SLOs and limits total execution duration;
- **Money budget:** Keeps unit economics compatible with workload value;
- **Search budget:** Limits breadth, depth, or the total number of candidates explored.

These budgets represent maximum limits, not consumption targets.

The fact that a task is authorized to use twenty steps does not mean that it should execute all twenty. If the verifier confirms success after three, continuing execution only reduces efficiency.

17.4.3. Termination and Escalation

Stop conditions need to be explicit. An execution may terminate under different circumstances:

- **The verifier confirms success:** There is no justification for continuing to consume compute;
- **Marginal gain disappears:** Additional iterations stop producing new evidence or improvements;
- **The budget is reached:** The runtime needs to stop, degrade gracefully, or escalate;
- **Progress stops occurring:** Loops need to be detected even before the maximum budget is reached;
- **Human review becomes mandatory:** Some cases must leave automation when risk or uncertainty exceeds defined limits.

Termination also needs to account for the opportunity cost of the capacity being consumed. A single deep execution may compete for accelerators or services required by dozens of simple tasks.

The principle is: **budgets define how much the system may consume; termination determines when continuing to consume no longer makes sense.**

17.5. Boundaries Between Reasoning, Retrieval, Tools, and Runtime

Reasoning, retrieval, and tools solve different classes of problems. Mixing them as though they were manifestations of the same capability tends to produce fragile and economically inefficient architectures.

17.5.1. Retrieval Provides Information

Retrieval is appropriate when the system needs knowledge that should not depend exclusively on model parameters. This occurs primarily in several situations:

- **Knowledge changes frequently:** The response needs to use an up-to-date source;
- **Provenance is important:** The system needs to demonstrate where a particular claim came from;
- **The corpus exceeds the context window:** Search needs to select only the relevant information;
- **Information is proprietary or user-specific:** The model does not possess this knowledge by default.

Additional reasoning does not correct the absence of this information. If the evidence does not exist in context or in the available reliable parameters, “thinking more” does not create a source of truth.

17.5.2. Tools Execute Specialized Operations

Tools should be used when an implementation exists that can perform an operation with stronger guarantees than probabilistic generation. Important examples include:

- **Calculators for exact mathematics:** Remove unnecessary uncertainty from arithmetic;
- **Databases for transactional data:** Preserve an explicit source of truth;
- **Code execution for computation and experimentation:** Makes it possible to test hypotheses and verify solutions;
- **APIs for domain actions:** Connect the system to existing capabilities with defined contracts.

The model may decide which tool to use or how to interpret its result, but there is no advantage in replacing deterministic operations with generated text when a specialized component already solves the problem more accurately.

17.5.3. Reasoning Composes Decisions

Reasoning should be concentrated at points where interpretation, decomposition, choice, or synthesis add value. Appropriate responsibilities include:

- **Decomposing a problem:** Determining which subtasks need to be solved;
- **Choosing sources or tools:** Deciding which mechanism can produce the required evidence;
- **Comparing alternatives:** Evaluating trade-offs among different solutions;
- **Synthesizing evidence:** Transforming multiple observations into a coherent decision;
- **Adapting strategy:** Modifying the plan when new information changes assumptions.

A useful architectural pattern is: **retrieve → reason → act → verify.** The boundaries do not need to correspond literally to four independent services, but their responsibilities should remain clear.

This separation improves troubleshooting. When a task fails, it becomes possible to determine whether the problem occurred because information was missing, because reasoning misused the evidence, because a tool failed, or because the verifier approved an inadequate output.

17.5.4. Reasoning as a Distributed Workflow

As the number of steps increases, a simple model call may become a distributed workflow. At this point, traditional Software Engineering and Distributed Systems principles become fundamental:

- **Reasoning state needs to be explicit:** Plans, evidence, results, and completed actions should not depend solely on implicit state within a sequence of messages;
- **Checkpoints enable recovery:** Long-running workflows need to survive process crashes, deployments, and interruptions;
- **Idempotency protects retries:** Repeating a tool call must not duplicate payments, messages, or other critical operations;
- **Timeouts need to exist locally and globally:** A slow external dependency should not block the workflow indefinitely;
- **Durable execution may be required:** Tasks lasting minutes or hours should not depend on a single HTTP connection remaining alive;
- **Bulkheads reduce noisy neighbors:** Deep reasoning may require independent resource pools to avoid degrading interactive traffic;
- **State transitions need to be observable:** It must be possible to reconstruct how execution progressed without relying exclusively on a textual narrative generated by the model.

This change in perspective is important. When reasoning grows beyond a single inference, the system stops being merely an application that “calls an LLM” and begins operating as a **distributed workflow runtime with probabilistic components**. This means retries, queues, concurrency limits, persistence, recovery, idempotency, and isolation become as important as prompt design.

17.6. Operating Reasoning Systems in Production

Reasoning systems need to be evaluated and operated from a systemic perspective. Task success remains fundamental, but observing only the final response hides how much work, risk, and cost were required to produce it.

17.6.1. Observability and Evaluation

A mature platform measures both outcome and process. Several metrics are particularly useful:

- **Task success:** Measures whether the final objective of the task was achieved;
- **First-pass success:** Shows how many cases are resolved before retries, escalation, or refinement;
- **Recovery rate:** Measures how many initial failures are recovered through verification, retry, or escalation;
- **Steps per task:** Exposes growth in complexity and may reveal planner regressions or loops;
- **Tool calls per task:** Helps identify excessive or unnecessary use of external dependencies;
- **Verification failure rate:** May reveal deterioration before final task success declines, especially when retries are still masking the problem;
- **Escalation rate:** Indicates whether routing and model tiers are appropriately sized;
- **Loop rate:** Measures executions that stop making progress;
- **Cost per successful task:** Directly connects operation and quality;
- **Latency and cost percentiles:** p95 and p99 expose long tails hidden by averages.

These metrics need to be analyzed together.

Consider a system whose task success remains at 95%. At first glance, nothing has changed. However, if first-pass success has fallen from 90% to 60% and escalation rate has doubled, the platform may be preserving final quality through retries and expensive models. The aggregate metric hides economic and operational deterioration.

Evaluation should also separate different difficulty buckets:

- **Easy:** Cases that should require little capability;
- **Medium:** Intermediate workloads;
- **Hard:** Cases for which advanced reasoning should actually provide an advantage.

An evaluation dataset dominated by easy examples can hide regression precisely within the group that justifies the existence of reasoning. Another important practice is measuring **compute-quality curves**.

The question is: **how much does task success increase as the reasoning budget grows?**

This curve makes it possible to identify the point of diminishing returns. The first increment in compute may produce a large quality increase, while the tenth may significantly increase cost while only marginally improving success rate.

Ablations are also necessary. Verifiers, multiple sampling, search, planning, and other components need to justify their existence through experiments. If removing a complex layer barely changes task success, perhaps that layer should not be in production. Architectural complexity should exist because it produces measurable improvement, not because it appears sophisticated.

17.6.2. Security, Privacy, and Governance

Reasoning increases capability, but it also expands the attack surface. A model that only produces text has fewer opportunities to cause damage than a system capable of searching external sources, accessing internal information, executing code, and modifying systems through tools.

Controls need to evolve with this expansion:

- **External content must be treated as untrusted data:** Documents, pages, and tool outputs do not gain authority merely because they were added to the context;
- **Tool authorization must occur outside the model:** The LLM may propose an action, but identity, permissions, and policy determine whether it may be executed;
- **Tool arguments require semantic validation:** A payload may be syntactically correct and still represent a prohibited operation;
- **Budgets are also security controls:** Limits on tokens, steps, time, and tool calls reduce resource exhaustion and denial-of-wallet;
- **Data minimization reduces exposure:** The model should receive only the information necessary to perform its task;
- **Memory writes require greater rigor than temporary reasoning state:** Unverified hypotheses should not automatically become persistent memory;
- **Code execution requires isolation:** When reasoning can generate and execute code, the runtime needs to restrict access to the network, filesystem, and sensitive resources;
- **Auditability should record operational artifacts:** Model version, prompt or policy version, consulted sources, tools used, authorization decisions, and outputs need to be reconstructible.

One point is especially important: **auditability should not depend on the availability of a complete internal narrative of the model's processing.** For governance, what matters is recording inputs, evidence, actions, policy decisions, tool calls, versions, verification outcomes, and final outputs.

Reasoning capability increases flexibility. It should not automatically increase the authority granted to the system.

17.6.3. Performance and Capacity Planning

Reasoning workloads are highly heterogeneous. One request may terminate after a few seconds. Another may execute dozens of steps, retrieve documents, perform search, invoke tools, and use multiple models.

This distribution changes capacity planning:

- **QPS alone no longer represents load:** Tokens, reasoning effort, duration, and tool usage become necessary;
- **Tail workloads can dominate capacity:** A small fraction of deep reasoning tasks may consume a large portion of available compute;
- **Admission control can protect the system:** During saturation, lower-priority tasks may be rate-limited or queued;
- **Priority queues differentiate workloads:** Incident response may receive higher priority than background analysis;
- **Separate resource pools reduce noisy neighbors:** Interactive traffic and long-running jobs may have distinct capacity allocations;
- **Cost attribution by tenant and feature makes problems visible:** Without this separation, economically poor workflows remain hidden in aggregate metrics.

In highly variable systems, averages can be misleading. A workload with acceptable average cost may have a p99 high enough to compromise margin or capacity. Similarly, average latency may remain low while a fraction of users receives experiences incompatible with the product.

For reasoning workloads: **the distribution matters more than the average.**

17.6.4. Business Economics

Test-time compute only makes sense when it produces value proportional to the additional resources consumed. Economic analysis depends primarily on several variables:

- **Marginal value of higher correctness:** How much is increasing the probability of correctly solving the task worth?;
- **Cost of error:** A failure in brainstorming has a very different consequence from a failure in a financial or operational workflow;
- **Marginal cost of reasoning:** How many additional tokens, accelerator cycles, tool calls, and seconds are required?;
- **Cost of intervention:** How many failures still require human review?;
- **Impact on latency:** Quality improvements lose part of their value if the experience becomes excessively slow.

Different workloads therefore require different policies:

- **Low-risk tasks should remain inexpensive:** Deep reasoning is rarely economically justified for disposable outputs;
- **Critical tasks may justify strong verification:** Small quality improvements may avoid losses far greater than the cost of compute;
- **Premium reasoning can become part of the product:** Greater depth may be offered when users accept higher latency and price;
- **Cost per successful task should replace cost per request as the primary metric:** An expensive model may be economically superior if it reduces retries, failures, and human labor;
- **The quality-cost frontier must be measured:** The economically optimal point usually appears before the maximum theoretically achievable quality.

The objective is not to maximize intelligence per request. It is to maximize: **business value per unit of compute.**

17.7. Trade-Offs, Anti-Patterns, and Business Impact

Reasoning Models introduce a broad class of trade-offs. No configuration simultaneously maximizes quality, cost, latency, robustness, and operational simplicity. A Staff AI Engineer needs to make these compromises explicit.

The main ones recur consistently:

- **Quality vs. Compute:** More processing may increase task success, but consumes more resources;
- **Quality vs. Latency:** Deep reasoning may improve the response while simultaneously making the experience unsuitable for interactive workloads;
- **Breadth vs. Depth:** Exploring many hypotheses reduces premature commitment, while deeply investigating one trajectory may be required for long problems;
- **Single Sample vs. Multiple Samples:** Diversity increases the probability of finding a good solution, but multiplies inference cost;
- **Generation vs. Verification:** Verification increases reliability, but introduces another stage that also has cost and an error rate;
- **Self-Correction vs. Independent Verification:** Model self-review is convenient, while independent evidence tends to provide greater confidence;
- **Fixed vs. Dynamic Budget:** Fixed budgets improve predictability, while adaptive compute improves efficiency in heterogeneous workloads;

- **Small Model + Reasoning vs. Large Model:** Part of the capability may come from search and test-time compute instead of permanently increasing model size;
- **Reasoning vs. Retrieval:** More inference does not solve missing, stale, or unauthorized knowledge;
- **Reasoning vs. Deterministic Tools:** Reasoning offers flexibility, while specialized tools provide stronger guarantees for well-defined operations;
- **Autonomy vs. Control:** Greater freedom increases capability as well as blast radius;
- **Synchronous vs. Asynchronous Execution:** Interactive interfaces require smaller budgets, while asynchronous jobs can explore deeper trajectories;
- **Maximum Quality vs. Business ROI:** The last increment in accuracy may cost more than the economic value it produces.

The Staff-level responsibility is not to eliminate these trade-offs. It is to convert them into explicit policies, measurable metrics, and decisions compatible with each workload.

17.7.1. Anti-Patterns

Several patterns indicate that the architecture is using reasoning as a substitute for engineering. They should be recognized quickly during design reviews:

- **Deep reasoning on every request:** Using maximum capability for trivial tasks increases cost and latency without proportional value;
- **More tokens as the automatic response to low quality:** The bottleneck may be retrieval, data, model capability, or tools;
- **Reasoning to replace deterministic software:** Asking the model to calculate or validate something traditional software can execute exactly adds error;
- **Self-correction as the only defense:** The same model may repeat or reinforce the original failure mode;
- **Multiple sampling without a strong selection mechanism:** More candidates simply multiply cost if the system cannot recognize the best one;
- **Search without pruning and budgets:** The alternative space may grow combinatorially;
- **Loops without progress detection:** Repeating actions without new evidence turns reasoning into waste;
- **Allowing the model to control unlimited resources:** Budgets need to be imposed by the runtime;
- **Persisting unverified hypotheses as memory:** Incorrect temporary state can contaminate future executions;
- **Measuring only final accuracy:** This metric hides retries, tool calls, escalations, latency, and cost;
- **Mixing deep reasoning and interactive traffic without isolation:** Heavy tasks can degrade the experience for every user;
- **Delegating authorization to the model:** Reasoning may select a candidate action, but it must never create permission to execute it;
- **Using a planner when the workflow is already known:** This introduces variability where a deterministic process would be safer;
- **Confusing a convincing explanation with correctness:** Narrative coherence is not a substitute for evidence.

The common pattern behind these anti-patterns is the use of probabilistic capability to replace boundaries that should remain explicit.

17.7.2. Business Impact

When properly designed, Reasoning Models can increase the amount of intellectually complex work that becomes economically automatable. The primary potential benefits include:

- **Higher task success on complex problems:** Coding, debugging, analysis, and planning may require less human intervention;
- **Reduced rework:** Verification can detect failures before they reach users or downstream systems;
- **Higher specialist productivity:** The system can assist with tasks requiring deeper composition and investigation;
- **Better compute allocation:** Adaptive reasoning avoids paying for maximum capability on every interaction;
- **Greater automation of composite workflows:** Planning, tools, and verification make it possible to execute processes that previously depended on human coordination.

These benefits, however, introduce new costs:

- **Inference becomes more expensive:** A multi-step task may consume several times the resources of direct generation;
- **Latency increases:** Additional quality may stop providing value when users do not accept the required time;
- **Engineering burden increases:** State, retries, queues, durable execution, observability, and security become more important;
- **Autonomy increases risk:** An incorrect action may produce far greater consequences than an incorrect textual response;
- **Capacity planning becomes more difficult:** Per-task variability increases the importance of long-tail management.

The fundamental relationship is: **test-time compute turns quality into an economic variable controllable per request.** This property is powerful because it makes it possible to spend more precisely where marginal value is higher. At the same time, it directly connects Reasoning Models to FinOps, SRE, Product Engineering, and governance.

Chapter Summary

Reasoning Models make it possible to exchange additional compute for a higher probability of solving certain classes of difficult problems. However, this mechanism produces an advantage only when the architecture can transform additional compute into useful exploration, better evidence, or greater reliability.

The main principles can be summarized as follows:

- **Treat reasoning as search when the problem requires exploration:** Difficult cases may require multiple alternatives rather than direct generation;
- **Use decomposition only when it reduces complexity:** Every step needs to make part of the problem simpler or more verifiable;
- **Plan without making the plan rigid:** Replanning needs to occur when new evidence invalidates assumptions;
- **Prefer deterministic verification whenever possible:** Code, schemas, calculations, invariants, and databases produce stronger evidence than probabilistic reflection;
- **Use self-correction as refinement, not as a guarantee:** Critical correctness requires independent evidence when available;

- **Use multiple sampling only when diversity and selection justify the cost:** More samples do not automatically imply higher quality;
- **Impose pruning and budgets on search:** Exploration needs explicit limits;
- **Allocate compute dynamically:** Simple, difficult, and critical cases should not necessarily receive the same resources;
- **Define budgets for tokens, steps, tool calls, time, and money:** Resource control is part of system architecture;
- **Separate reasoning, retrieval, and tools:** Each mechanism solves a different class of problem;
- **Use routing and cascades:** Expensive capability should be concentrated on tasks that actually require it;
- **Design long-running reasoning as a distributed workflow:** State, checkpoints, idempotency, timeouts, and recovery become necessary;
- **Monitor the process, not only the outcome:** First-pass success, recovery, escalation, loops, and cost per success reveal real efficiency;
- **Keep authorization and policies outside the model:** Greater reasoning capability does not imply greater authority;
- **Optimize business value per unit of compute:** The best architecture is not the one that reasons the most, but the one that uses additional processing only where it materially changes the outcome.

The primary lesson is that Reasoning Models do not eliminate the foundations of AI Engineering. They make those foundations even more important.

An inference that could previously be represented as a single call begins to operate as an adaptive system capable of retrieving information, decomposing problems, searching alternatives, using tools, revising hypotheses, and verifying results. The greater this capability becomes, the greater the discipline required around state, budgets, authorization, reliability, observability, and economics.

Practical Framework for Designing Systems with Reasoning Models

When analyzing an application that apparently requires advanced reasoning, a Staff AI Engineer should begin by determining whether reasoning is actually the correct mechanism.

The first step is to determine the nature of the task:

- **Complexity:** Does the solution truly require multiple inference steps?;
- **Ambiguity:** Are there enough alternatives to justify exploration?;
- **Risk:** What is the cost of producing an incorrect response?;
- **Baseline:** Does a direct response from a smaller model already provide sufficient quality?;
- **Determinism:** Would an explicit workflow solve the problem with greater predictability?

The purpose of this stage is to prevent reasoning from being introduced merely because it is available. Next, identify the actual bottleneck:

- **Knowledge:** Does the model have all the required information?;
- **Retrieval:** Is there missing or outdated information that should be retrieved?;
- **Tools:** Should any operation be delegated to deterministic software?;
- **Model capability:** Is the problem truly reasoning, or does the model simply lack sufficient capability?;
- **Architecture:** Is test-time compute being used to compensate for an upstream deficiency?

This diagnosis determines which investment provides the greatest return. Next, evaluate decomposition and planning:

- **Subproblems:** Which parts can be solved independently?;
- **Coordination cost:** Does decomposition reduce complexity or merely create more state?;
- **Verification boundaries:** Which intermediate outputs must be validated before continuing?;
- **Planning horizon:** How far into the future can the system plan using reliable assumptions?;
- **Replanning events:** Which changes should invalidate the current plan?

The objective is not to maximize steps. It is to find the smallest structure capable of making the problem sufficiently verifiable. Next, design verification:

- **Deterministic verifier:** Is there an objective way to test the solution?;
- **Source of truth:** Can any result be checked again against a database, API, or other system?;
- **Schemas and invariants:** Are there automatically verifiable properties?;
- **Local verification:** Can intermediate errors be stopped before they propagate?;
- **Model-based verifier:** If necessary, what is its error rate, and how correlated is it with the generator?

The stronger the verifier, the greater the potential value of test-time compute tends to be. Next, evaluate whether multiple trajectories add value:

- **Single-sample baseline:** Does one attempt already provide sufficient quality?;
- **Diversity:** Do additional samples produce materially different solutions?;
- **Marginal gain:** How much does the second, fourth, or tenth attempt actually increase task success?;
- **Selection:** Is there a scorer capable of recognizing the best solution?;
- **Correlation:** Does agreement represent evidence or merely shared systematic error?

Multiple sampling should exist because it changes the success distribution, not simply because it increases the number of outputs. If search over reasoning paths is used, explicitly define the search space:

- **Breadth:** How many branches may remain active?;
- **Depth:** What is the maximum depth?;
- **Pruning:** How will unpromising trajectories be eliminated?;
- **False pruning:** What is the risk of prematurely discarding a valid solution?;
- **Search budget:** How much compute can be consumed before termination?

No search architecture should have effectively unbounded growth. Next, define adaptive compute and routing:

- **Difficulty signals:** How does the system estimate that a task requires greater capability?;
- **Early success:** What evidence allows execution to terminate within the inexpensive tier?;
- **Escalation:** Should verification failure increase the budget, switch the model, or require another mechanism?;
- **Risk sensitivity:** Should the cost of error change the allowed compute?;
- **Model tiers:** Which workloads belong to small, general-purpose, or advanced reasoning models?;
- **Under-routing and over-routing:** What is the cost of making a mistake in either direction?

Routing needs to be evaluated as part of the system rather than as an isolated classifier. Next, impose resource budgets and termination:

- **Steps:** What is the maximum number of cycles?;
- **Tokens:** What textual consumption is allowed?;

- **Tools:** How many external calls may occur?;
- **Wall-clock time:** How long may the workflow remain executing?;
- **Money:** What is the maximum economically acceptable cost per task?;
- **Termination:** What evidence confirms that continued reasoning will not produce sufficient return?;
- **Human escalation:** When does automation need to stop?

None of these limits should depend exclusively on the model's willingness to terminate its own execution. Next, analyze runtime and reliability:

- **Execution model:** Is the task synchronous, or does it require asynchronous processing?;
- **State persistence:** Does the workflow need to survive process failures?;
- **Checkpoints:** Which points enable recovery?;
- **Idempotency:** Can retries duplicate side effects?;
- **Timeouts:** Are there limits for individual dependencies and for total execution?;
- **Resource isolation:** Does deep reasoning need to remain separated from interactive traffic?;
- **Noisy neighbors:** Can a single execution consume enough resources to degrade service for other users?

At this stage, reasoning needs to be treated as a distributed workload. Next, define observability and evaluation:

- **Task success:** Does the system solve the final problem?;
- **First-pass success:** How many cases require recovery?;
- **Recovery rate:** How many initial failures are recovered?;
- **Steps and tool calls:** How much work is executed per task?;
- **Verification failures:** How dependent is the system on retries?;
- **Escalation rate:** Is routing appropriately calibrated?;
- **Loop rate:** How many executions stop making progress?;
- **Percentiles:** What are p50, p95, and p99 for latency, tokens, and cost?;
- **Compute-quality curve:** How much additional success does each budget increment produce?

Evaluation should indicate not only whether the architecture works, but whether it works efficiently. Next, evaluate robustness and security:

- **Perturbation robustness:** Do small irrelevant changes alter the conclusion?;
- **Contradictory evidence:** Does the system abandon hypotheses when contrary evidence appears?;
- **Falsification:** Is there a mechanism for searching for evidence against the current conclusion?;
- **Untrusted inputs:** Which sources may contain malicious content?;
- **Tool authorization:** Who decides whether an action is permitted?;
- **Argument validation:** Are tool calls validated beyond the schema?;
- **Resource abuse:** Can an adversarial request induce excessive reasoning?;
- **Sandboxing:** Does code execution have sufficient isolation?;
- **Data minimization:** Does the model receive only the necessary data?

Reasoning expands capability and therefore needs to be accompanied by stronger boundaries. Next, evaluate memory, governance, and auditability:

- **Persistence:** Which information actually needs to survive the task?;
- **Memory verification:** Can a hypothesis be persisted without validation?;
- **Provenance:** Does persistent information preserve its origin?;
- **Versioning:** Can the model, prompts, and policies used be identified?;
- **Tool traceability:** Can executed actions be reconstructed?;

- **Policy decisions:** Is it possible to determine why a particular action was authorized or blocked?;
- **Incident reconstruction:** Is there sufficient operational evidence to analyze a problematic execution?

The objective is to achieve auditability through system artifacts rather than relying on an informal narrative of reasoning. Finally, connect the entire architecture to economics and business value:

- **Cost per successful task:** How much does it cost to produce a valid result?;
- **Human review:** How much manual work still remains?;
- **Error economics:** What is the economic value of preventing a failure?;
- **Latency tolerance:** How much additional time will the user accept?;
- **Diminishing returns:** Where is the point at which additional reasoning stops producing sufficient return?;
- **Architecture alternatives:** Would a larger model with less reasoning be better? Would a smaller model with more reasoning have better economics? Would a deterministic solution be significantly less expensive?;
- **Business KPI:** Which business metric improves as a consequence of this capability?;
- **Cycle time:** How much operational time is reduced?;
- **Rework:** How much repeated work is eliminated?;
- **Risk reduction:** What financial, operational, or security exposure is reduced?;
- **Engineering leverage:** Can the infrastructure created benefit other products?;
- **Complexity test:** If half of the reasoning architecture were removed, would the product continue delivering nearly the same value?

This last question is particularly important.

The mindset expected from a Staff AI Engineer is not to evaluate a Reasoning Model by the number of steps it executes, the length of its response, or the apparent sophistication of its workflow. The objective is to design a system in which every increment of decomposition, planning, search, verification, routing, and test-time compute has an explicit function, is observable, and produces sufficient return in quality, reliability, risk reduction, or business impact to justify its cost.

In production, reasoning is not a competition to determine how much compute a system can consume. It is an engineering problem about **where additional compute materially changes the outcome and where it merely increases complexity**.