

Agentic AI

Architecture, Theory, and Practice

Marco Dinacci

Selected pages from four chapters, including diagrams, a Python case study, and the book's approach to agent design.

First edition

21-page preview

A guide to the sample

These excerpts reproduce the published layout. Original page numbers are retained; gaps between chapters are intentional. Each excerpt stops at a section boundary. The full book contains fourteen chapters.

CHAPTER 01 / BOOK PAGES 2-4

From Programs to Agents

How control flow changes when software delegates next-step decisions.

CHAPTER 03 / BOOK PAGES 58-64

Retrieval and Context

Context as a view over state, a retrieval pipeline, and a repository-repair case study.

CHAPTER 04 / BOOK PAGES 73-78

Tools and Runtime Boundaries

Why tools are ordinary APIs and how the runtime turns model proposals into actions.

CHAPTER 07 / BOOK PAGES 173-175

When to Write an Explicit Plan

The boundary between transient model reasoning and durable application plans.

1. Foundations

1.1. From Programs to Agents

Every significant shift in software engineering has begun with a deceptively simple idea. The transition from assembly language to high-level programming languages changed the way we expressed computation. Object-oriented programming encouraged us to organize software around interacting objects rather than procedures. The internet transformed applications from isolated executables into distributed systems. Cloud computing redefined infrastructure as a service rather than a physical asset. Each of these developments introduced new technologies, but their lasting impact came from something deeper. They changed the way software was conceived. Once the underlying abstraction changed, everything else—frameworks, tools, programming languages, and design patterns—evolved around it.

The emergence of AI agents represents another such transition. Recent advances are often interpreted primarily as advances in language models. After all, the most visible breakthrough has been the extraordinary ability of modern models to write code, answer questions, summarize documents, and engage in conversations that would have seemed almost impossible only a few years ago. From this perspective, AI appears to be a new capability that can be added to existing applications much like a database, a search engine, or a machine learning model.

There is certainly some truth in that observation. Many successful applications today use language models in exactly this way. They translate text, classify documents, extract structured information, or generate reports. The surrounding software remains unchanged; the model replaces a deterministic algorithm with a statistical one. Useful as these applications are, they do not fully explain why the industry has become so interested in agents. The excitement surrounding agentic systems is not primarily about producing better text. It is about a different way of organizing computation. For most of the history of software engineering, programs have been passive.

A program waits for an external event—a button click, a Hypertext Transfer Protocol (HTTP) request, a scheduled job, or a message arriving on a queue. Once activated, it executes a sequence of operations that has already been determined by its developer. The program may contain loops, conditional branches, recursion, concurrency, and sophisticated algorithms, yet one principle remains constant. The programmer has already decided every meaningful aspect of the program’s behavior before it begins executing. This is so familiar that it often goes unnoticed.

1. Foundations

Consider a web application processing an online purchase. A customer submits an order. The application validates the request, verifies payment, updates inventory, generates a shipment, and sends a confirmation email. Every one of these operations may involve thousands of lines of code and interactions with dozens of external services, but the overall structure of the computation has already been established. The application is following a path that someone designed long before the customer ever placed the order. Even software that appears highly dynamic follows the same principle. A chess engine evaluates millions of possible moves, yet the algorithm governing that search was written by its developers. A compiler chooses among numerous optimization strategies, but the logic determining those choices exists in the source code. A distributed scheduler dynamically allocates work across thousands of machines, but the scheduling policies themselves are still encoded by engineers.

Traditional software can therefore be understood as a system whose behavior is determined by explicit instructions written in advance. This view has shaped nearly every discipline within software engineering. Algorithms describe procedures. Workflow engines describe sequences of activities. Finite-state machines describe transitions between known states. Business rules encode organizational policies. Even machine learning systems are typically embedded inside deterministic pipelines that specify precisely when the model should be invoked and how its predictions should be interpreted. The common thread is not the absence of complexity. It is the ownership of control.

The developer decides what the software is allowed to do, and the computer executes those decisions. This arrangement has served the industry well. Deterministic control flow makes software predictable. Execution paths can be inspected before deployment, tested systematically, optimized for performance, and reasoned about with considerable confidence. When something goes wrong, engineers can usually trace the failure back to a particular function, a specific condition, or an incorrect assumption encoded in the program itself. The strengths of traditional software are inseparable from this model of execution. Its limitations are as well.

As applications began tackling increasingly open-ended problems, developers found themselves attempting to encode behavior that was inherently difficult to describe as deterministic logic. A customer support system must select the document that best answers an unfamiliar question. A coding assistant must choose which file to inspect first in a repository it has never seen. A research assistant must determine whether the available evidence is sufficient or another search is warranted. These problems do not yield a single algorithm; they require judgment.

For many years, software addressed them by accumulating increasingly elaborate heuristics. Rules were added. Exceptions were introduced. Decision trees became larger. Configuration files became more complicated. Each improvement solved a handful of additional cases while making the overall system progressively more difficult to understand and maintain. Language models offered a different possibility. Rather than attempting to encode every decision explicitly, we could delegate some of those decisions to a learned policy capable of adapting to situations that had not been anticipated during development. That idea is far more significant than it may initially appear.

1. Foundations

It does not merely change how we implement software. It changes who determines what happens next. This book is about the consequences of that shift. It is not a book about prompt engineering, nor is it a catalog of frameworks or APIs. Those technologies are important, but they are expressions of a deeper architectural change. The central question we will explore is how software must evolve once it is no longer required to have every significant decision encoded before execution begins. To answer that question, we must first understand precisely what has changed.

The next section shows that the defining characteristic of modern agents is not that they use language models. It is that they introduce a new relationship between software and control flow—one in which certain decisions are no longer made exclusively by the programmer, but emerge dynamically as the system interacts with its environment.

3. Retrieval and Retrieval-Augmented Generation (RAG)

3.1. Stateless Inference, Stateful Execution

A language model begins each inference with the information supplied to it at that moment. It does not carry an intrinsic record of the previous call, the user's earlier decisions, or the tool result that changed the state of the task. When an assistant appears to remember, software outside the model has preserved information and presented an appropriate portion of it again. Continuity is therefore a property of the runtime, not of inference itself.

This distinction matters because an agent is a computation extended across time. A single model call may classify a document or draft a response, but an agent observes, decides, acts, and then reasons over the consequences. Each iteration depends on state produced by earlier iterations: the objective, completed actions, unresolved questions, tool outputs, policy constraints, and changes in the environment. Without that state, the model can produce another plausible response, but it cannot reliably continue the same computation.

The runtime creates continuity by maintaining an execution state that outlives any one inference. Before calling the model, it assembles a bounded view of that state. After the model responds, it validates the proposed action, executes it when permitted, records the result, and updates the state for the next iteration. Some state is transient and useful only during the current task. Other information may remain valuable across tasks and therefore deserves longer retention. Both forms belong to the architecture surrounding the model.

Memory, in this sense, is not a magical faculty added to a language model. It is the set of policies and mechanisms through which past information is allowed to influence future decisions. Storage is necessary, but insufficient. A system can preserve every event it has ever observed and still fail to remember what matters. The engineering problem is to select the information that should affect the next decision while excluding information that is stale, irrelevant, untrusted, or too expensive to include.

3.2. Context Is a View, Not a Memory Store

The context window is the bounded input to one inference. Working memory is the runtime-managed state needed to perform the current task. Long-term memory contains information

3. *Retrieval and Retrieval-Augmented Generation (RAG)*

expected to remain useful across tasks. These concepts are related, but they are not interchangeable. At any moment, the runtime may hold far more working state than the model can or should see. A coding agent may have inspected hundreds of files, executed dozens of commands, and accumulated a large trace of intermediate conclusions. Replaying that trace verbatim would consume tokens, obscure current priorities, and preserve mistakes long after they ceased to be useful. Instead, the runtime constructs an inference context from selected instructions, current goals, recent observations, summaries, retrieved knowledge, tool descriptions, and relevant constraints.

Context should therefore be understood as a view over state. It is assembled for a particular decision and may change from one inference to the next even when the underlying execution state remains stable. A planner deciding which repository area to inspect needs architectural summaries and recent search results. A tool-selection step needs available capabilities and argument constraints. An evaluator needs the candidate artifact, acceptance criteria, and evidence produced during execution. Supplying every component with the same context is both wasteful and a source of accidental coupling.

3. Retrieval and Retrieval-Augmented Generation (RAG)

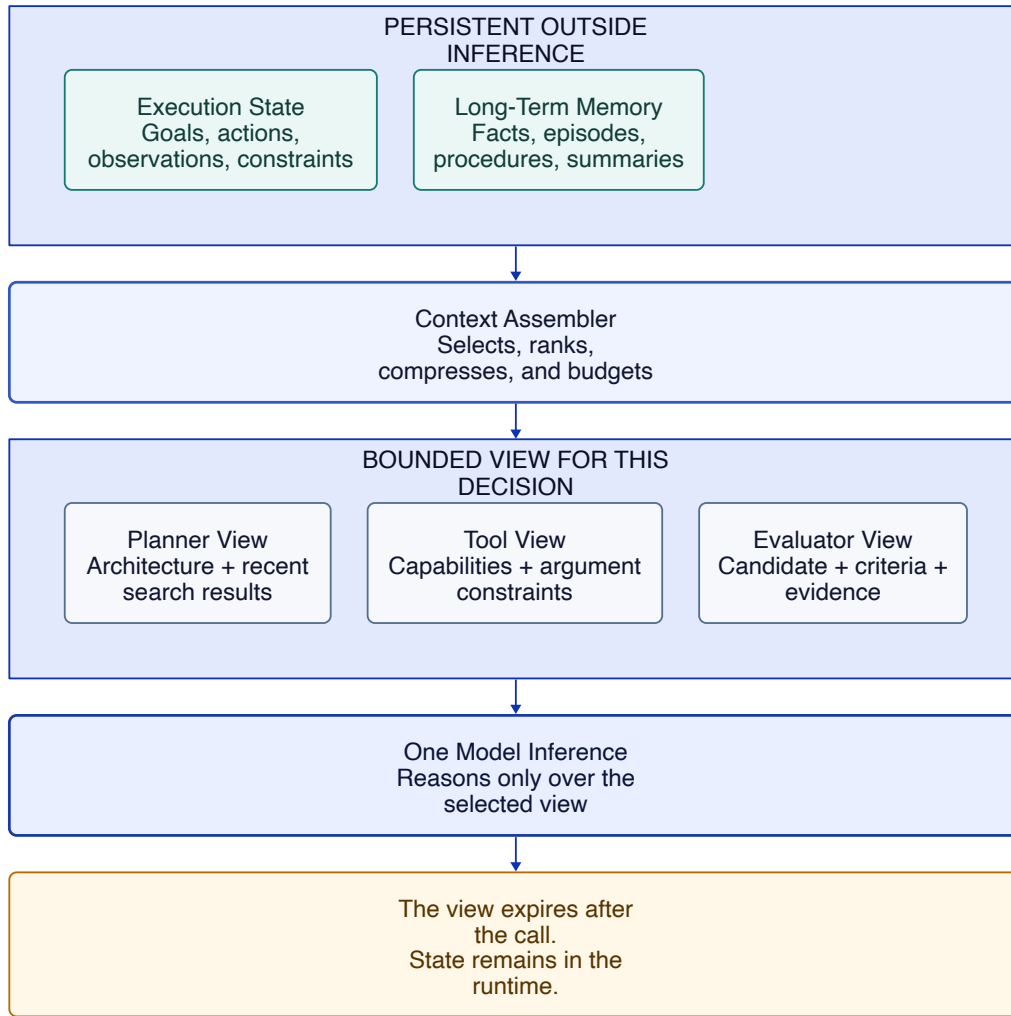


Figure 3.1.: Context is a bounded, decision-specific view assembled from state that persists outside the model inference.

This view-based model also clarifies responsibility. The model reasons only over what it receives. The runtime decides what is eligible, how it is represented, which sources are trusted, how competing information is ranked, and how much of the token budget each category may consume. Poor context assembly can make a capable model appear confused; disciplined assembly can improve performance without changing the model at all.

3.2.1. Progressive Loading and Compaction

Context assembly is a continuing process, often called **context engineering**. At each decision, the harness selects instructions, tool descriptions, retrieved evidence, recent results, and any

3. Retrieval and Retrieval-Augmented Generation (RAG)

summaries needed to continue. Loading all available information at the start wastes capacity and can obscure the constraints that matter now. Progressive loading exposes compact descriptions first, then loads full instructions or resources when a task requires them. Chapter 8 applies this pattern to reusable skills.

Large intermediate results should normally remain in an external artifact, with a bounded excerpt and a reference in context. A repository search may produce thousands of matches; the model can receive a count, relevant matches, and an artifact identifier for later inspection. The reference must preserve scope and source version. Otherwise a later read may silently retrieve different content, or an artifact created for one tenant may become available to another. Truncation should be explicit so the model does not mistake an excerpt for a complete result.

Compaction replaces selected conversation history with a shorter representation. A useful handover preserves the objective, acceptance criteria, unresolved questions, consequential decisions, current constraints, and references to supporting evidence. It should distinguish observations from hypotheses and record incomplete work. It is a lossy view of execution, not an authoritative replacement for task state or original artifacts. Critical values such as approval status and completed side effects belong in structured state that can be checked independently of a summary.

Compaction policies should be evaluated by whether a fresh inference can continue correctly, rather than by compression ratio alone. Test whether the agent retains constraints, avoids repeating completed actions, and can recover the evidence behind a conclusion. The same principle applies when delegating to another agent: transfer the context needed for its decision, with source and authority preserved, rather than copying the entire conversation by default.

3.3. Retrieval as Context Selection

Retrieval is the process of selecting stored information for possible inclusion in an inference context. The word *possible* is important. Finding an item does not imply that it should be shown to the model, and similarity does not imply usefulness. A production retrieval system usually performs several decisions: identify the information need, search eligible sources, score candidates, filter them by policy and metadata, rerank them for the current task, and fit the best evidence into a limited context budget.

This is why retrieval is more than a database query. Databases answer requests expressed against a known schema. An agent often begins with an underspecified problem and must infer what evidence would reduce uncertainty. A failed test may call for source code, recent commits, build instructions, or an earlier incident report. The literal wording of the user's request may mention none of them. Query construction is therefore part of reasoning, while execution of the query remains a deterministic runtime operation.

The retrieval pipeline should keep these responsibilities separate. The planner can state an information need and propose search terms or constraints. The runtime enforces access boundaries, executes searches, combines lexical, semantic, graph, or structured retrieval, and records

3. Retrieval and Retrieval-Augmented Generation (RAG)

provenance. A reranker estimates which results are likely to improve the present decision. A context assembler decides how much evidence to include and in what form. Keeping these stages explicit makes failures observable: engineers can distinguish a poor query from a missing document, a ranking error, an authorization filter, or a context-budget decision.

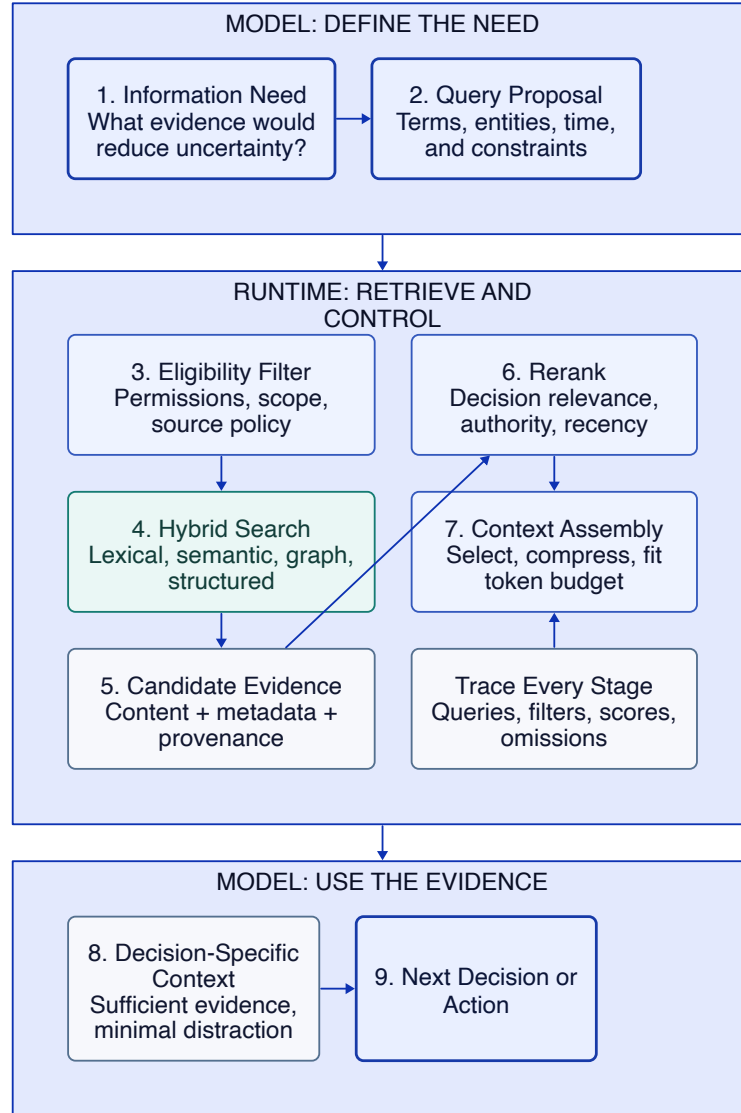


Figure 3.2.: A production retrieval pipeline separates reasoning about the information need from controlled search, ranking, context assembly, and evidence use.

The objective is not maximum recall. Returning every potentially related item transfers the selection problem into the model's context, where it becomes more expensive and less controllable. Good retrieval seeks sufficient evidence with minimal distraction. It should make the

next decision better, not merely make the prompt longer.

3.4. Case Study: Preparing a Repository-Repair Context

Consider an agent investigating a failed OAuth callback after a deployment. Its execution state may contain a build log, several source files, deployment metadata, older incident reports, and a large repository index. The next inference does not require all of it. It requires a bounded set of attributable evidence relevant to the failure: the callback implementation, the current configuration, and perhaps the recent deployment change.

The application retains ownership of the repository index and its access policy. It supplies a retriever that returns scored, attributable results; `bbagents` then turns the selected evidence into a budgeted context rather than treating retrieval as a memory dump.

```
from bbagents.context import ContextBudget, ContextBuilder
from bbagents.retrieval import RetrievalAugmenter
```

```
retrieval = RetrievalAugmenter(
    retriever=repository_index,
    context_builder=ContextBuilder(
        ContextBudget(max_items=3, max_characters=3_000)
    ),
)
repair_context = retrieval.augment(
    "OAuth callback rejected after deployment", limit=8
)
```

The index may return eight candidates, but the model receives at most three that fit the context budget. Each retained item carries its source, score, metadata, and provenance, so a later trace can explain why it was admitted. The resulting context can then be used for a bounded decision such as identifying the next inspection step.

```
from dataclasses import dataclass

from bbagents.decision import DecisionRequest

@dataclass(frozen=True)
class InspectionStep:
    path: str
    reason: str
```

3. Retrieval and Retrieval-Augmented Generation (RAG)

```
request = DecisionRequest(  
    task=(  
        "Identify the next repository inspection needed to diagnose "  
        "the callback failure."  
    ),  
    system="Base the recommendation only on the retrieved evidence.",  
    context=repair_context,  
    schema=InspectionStep,  
)
```

This division of responsibility is deliberate. Retrieval improves the information available to the model at one decision point; it neither changes the model's parameters nor creates durable memory by itself. The application can revise its index, ranking policy, source permissions, or context budget without changing the decision contract. That separation makes the retrieval path testable and preserves the distinction between persistent state, retrieved evidence, and the temporary view supplied to an inference.

4. Tools

4.1. Why Tools Exist

Language models infer from the information in their context; they cannot inspect the current state of a market, repository, database, or service unless the runtime retrieves it. Asking for Apple's current stock price illustrates the boundary. Training may provide financial knowledge, but only a market-data service can supply the present value. Without that observation, the model must refuse or guess.

Tools bridge this gap. They let the runtime obtain evidence, perform computations, and change external state. The model does not become smarter or acquire an API internally. It proposes an operation through a defined interface; deterministic software validates and executes that proposal, then returns the result as a new observation.

4. Tools

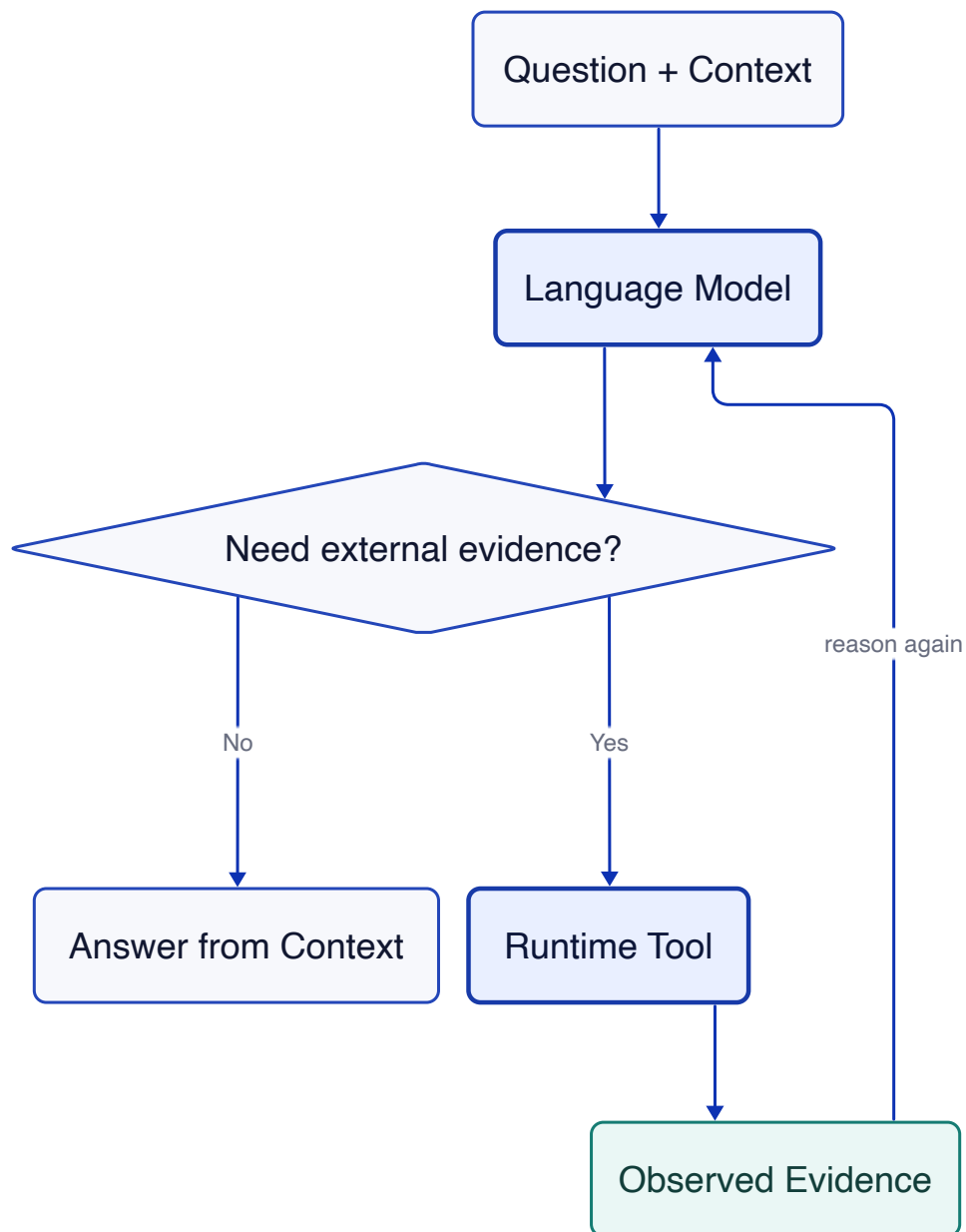


Figure 4.1.: Tools turn missing external facts into observations the model can reason over.

This creates the core agent loop: reason, act, observe, and reason again. A compiler can verify whether code builds. A repository API can count open pull requests. A network request can establish whether a service is available. In each case, the external system supplies facts that prediction alone cannot guarantee.

The runtime therefore becomes the agent's interface to reality. It authenticates requests, enforces

permissions, handles timeouts and failures, records effects, and decides what evidence returns to the model. The rest of this chapter develops that mechanism: tool schemas, function calling, execution loops, selection, concurrency, security, and orchestration.

4.2. A Tool Is Just an API

One of the unfortunate side effects of the recent popularity of agentic AI is that familiar software engineering concepts have acquired unfamiliar names. Few examples illustrate this better than the word *tool*. Reading current literature, one could easily conclude that tools represent an entirely new computational primitive invented specifically for language models, requiring specialized infrastructure and different design principles from conventional software. In reality, almost nothing could be further from the truth. A tool is software. More precisely, it is software that exposes a well-defined interface through which another component may request work. Software engineers have been designing such interfaces for decades. Every operating system exposes system calls. Every database exposes a query interface. Every web service exposes an API. Every library exports functions. Every command-line program accepts arguments, performs some computation, and returns a result.

From an architectural perspective, a language-model tool differs from these examples in only one respect. The caller happens to be a language model rather than another deterministic program. Recognizing this fact immediately removes much of the apparent mystery surrounding modern agent systems. When developers say that an agent “uses a tool,” they are not describing an exotic AI capability. They are describing a software component invoking another software component through a well-defined interface. The surrounding orchestration may be novel, but the underlying engineering principles are not. Figure 4.2 illustrates this relationship.

4. Tools

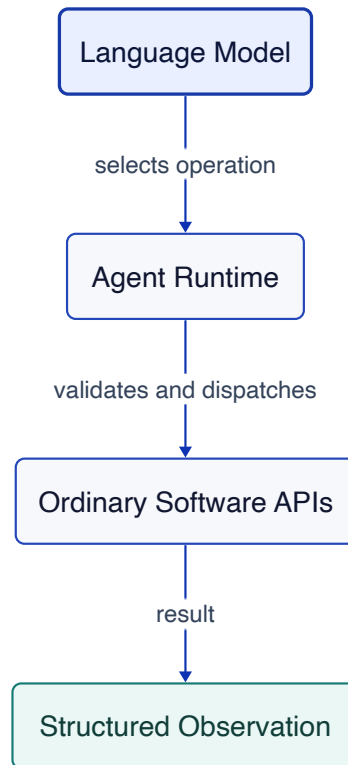


Figure 4.2.: The runtime translates model proposals into ordinary API calls and structured observations.

The language model never communicates directly with the database, the filesystem, or the network. Those systems neither understand natural language nor care about probabilistic reasoning. They expose deterministic interfaces designed according to ordinary software engineering principles. The runtime serves as the intermediary, translating the model's intentions into conventional API calls and translating the resulting observations back into a form suitable for the next inference. This translation layer is one of the defining responsibilities of every agent runtime. The comparison with a human developer using a software library clarifies its purpose. A developer retrieving all TODO comments from a repository might call a function such as

```
search_repository(pattern="TODO")
```

The compiler verifies that the function exists, validates the argument types, executes the implementation, and returns the result. At no point does the developer concern themselves with how directory traversal is implemented, how files are opened, or how regular expressions are evaluated. Those implementation details remain encapsulated behind the function's interface. An agent should interact with tools in exactly the same way. The language model should not reason about HTTP requests, Transmission Control Protocol (TCP) sockets, authentication headers, filesystem permissions, or SQL cursors. Those responsibilities belong to the runtime

4. Tools

and to the software implementing the tool. The model's role is to determine *which* operation should be performed and *why*. The runtime's responsibility is to determine *how* that operation is executed.

This separation permits the implementation to change without changing the model-facing abstraction. For example, we decide to replace a representational state transfer (REST) API with a gRPC service, migrate from PostgreSQL to another database engine, or introduce aggressive caching to improve performance. None of these changes should affect the language model. As long as the runtime continues exposing the same logical interface, the model's reasoning remains unchanged. The implementation may evolve substantially while the abstraction presented to the agent remains stable. This principle is identical to the one that governs good API design throughout software engineering. Applications should depend upon interfaces rather than implementations.

Agent architectures are no different. For this reason, tools should be designed with the same discipline that we apply to any other software interface. They should perform a single coherent operation, expose defined inputs, produce predictable outputs, and avoid hidden side effects whenever possible. A function named `do_everything()` would be considered poor API design in an ordinary software project, and it is equally poor tool design within an agent. A collection of focused operations such as

```
search_documents()
```

```
compile_project()
```

```
run_tests()
```

```
commit_changes()
```

is easier for human developers to understand and for the language model to reason about. The model must decide among the available actions before every tool invocation. Clear interfaces reduce ambiguity, while ambiguous interfaces increase the likelihood of incorrect tool selection. Inputs deserve particular attention because, unlike human developers, language models cannot inspect function signatures in the same way a compiler can. A Python definition such as

```
def search_documents(query: str):  
    ...
```

The signature is useful to a programmer but conveys relatively little semantic information to the model. The runtime therefore augments the interface with structured metadata describing the purpose of the operation, the meaning of each argument, and the constraints governing valid inputs. Later in the chapter we will examine JSON Schema in detail and see how modern providers use it to describe tools. For now, it is sufficient to recognize why these descriptions exist. The runtime is not merely exposing code. It is exposing the *meaning* of the interface in a form the language model can reason about.

4. Tools

The same observation applies to outputs. A tool that returns the string *Success* provides little information for subsequent reasoning because it does not identify the modified file, returned records, or passing tests. Effective tools return structured observations rather than conversational prose because structured data composes with subsequent computation. A compiler returns diagnostics, a database returns records, and a search engine returns documents; each result becomes part of the context assembled for the next inference. A tool is therefore not merely an executable function. It is a contract.

The language model promises to produce arguments conforming to a particular interface. The runtime promises to validate those arguments, execute the requested operation, handle failures appropriately, and return the resulting observation in a structured form. Neither side depends upon the internal implementation of the other. The model does not know whether the search tool queries Elasticsearch, PostgreSQL, or a vector database. The runtime does not know why the model decided that searching was appropriate. The contract between them consists solely of the interface itself.

Tool use is therefore an application of familiar interfaces rather than a new form of computation. The architectural innovation is allowing a probabilistic reasoning engine to orchestrate deterministic software through those interfaces. Because the model cannot execute Python code or issue HTTP requests directly, it describes the function that the runtime should execute. The next section examines how that description is generated and why modern providers have converged on similar mechanisms.

7. Planning

7.1. When an Agent Needs an Explicit Plan

The Observe–Reason–Act loop introduced earlier is sufficient for many agent tasks. A support agent may classify a request, retrieve one record, and draft a response. A coding assistant may inspect a compiler diagnostic, modify one file, and run the test suite. In these cases, the model can reason about the next action from the available context, and the runtime can validate the result before it has material consequences.

Some tasks have a different structure. Consider an agent assigned to migrate an application’s authentication system to OAuth. The work may require an inventory of existing session flows, an assessment of dependent services, a migration design, implementation changes, integration tests, a staged deployment, and a rollback plan. Later work depends on evidence produced by earlier work. A failed integration test may invalidate the proposed deployment sequence. The task may also need to survive a process restart, await approval, or transfer responsibility between agents.

Those properties justify an explicit application plan. An explicit plan is a runtime-owned representation of a goal, the work required to achieve it, the dependencies between that work, and the evidence required before execution may proceed. It is not a longer prompt and it is not a transcript of the model’s reasoning. It is application state.

7. Planning

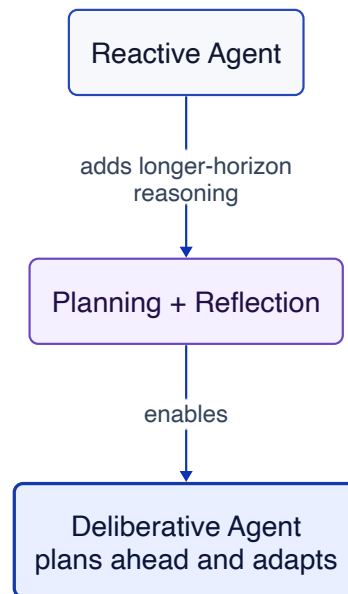


Figure 7.1.: Planning and reflection extend reactive execution into deliberative behavior

The distinction is practical. A reactive agent decides what to do next and then acts. An agent with an explicit plan also decides what to do next, but it does so within a visible structure that preserves the goal, completed work, blocked work, constraints, and evidence collected so far. The runtime can therefore determine which steps are ready, prevent invalid execution, record deviations, and resume the task after interruption.

Explicit planning is not a default requirement. It adds representation, scheduling, persistence, and evaluation work. The right question is not whether a language model can generate a sequence of steps; capable models routinely can. The question is whether the application needs that sequence to be durable, constrained, inspectable, or coordinated with other work.

7.2. Model Reasoning and Application Plans

The word *planning* describes two related but different activities. A language model may plan internally when it decomposes a request, compares possible actions, or decides which tool would provide the most useful evidence. This inference-time reasoning is useful, but it is transient. The model's internal reasoning is neither a durable workflow nor a source of authority over the runtime.

An application plan is an explicit artifact maintained outside the model. It identifies the goal, steps, dependencies, completion criteria, and execution state. The model may propose the plan, select a ready step, or recommend a revision after new evidence arrives. The runtime validates

7. Planning

and stores the resulting plan before it executes any action. The plan remains available even when the model changes, the process restarts, or a human reviews the work.

Model reasoning Interprets the current context and proposes a next action. It normally exists for one inference or a short reasoning episode.

Application plan Coordinates work across steps and time. It is typed state containing dependencies, policies, observations, and execution status for the lifetime of the task.

Authority The model proposes actions and revisions. The application plan defines what may execute and what evidence is required.

Inspection Model reasoning is available only when the application records an output. An application plan is inspectable, checkpointed, testable, and auditable.

The application should not attempt to reconstruct a plan from hidden model reasoning. It should ask the model for the structured information it needs: a proposed step, a dependency, a required observation, or a revised plan. Deterministic code should validate these proposals against the application's task model and policies.

This separation also clarifies when internal model planning is enough. A request can usually remain reactive when it has one bounded objective, few or no dependencies, inexpensive reversible actions, and a clear validation mechanism. A weather lookup, a document classification, or a single repository search normally does not require a durable plan. The model may reason about the appropriate tool internally, but the application need not represent that reasoning as a workflow.

An explicit plan becomes useful when work contains dependent stages, costly or irreversible actions, long-running execution, human approvals, delegation, or a requirement to resume and explain the computation. It is also useful when the system must compare plan quality across runs or maintain a budget for time, tool calls, and model inference. These conditions are architectural, not a measure of the task's intellectual difficulty. A simple deployment sequence may need an explicit plan because its effects are consequential, while a difficult one-shot analysis may not.