

MASSIMILIANO TARQUINI

BRICK BY BRICK

Building an Agentic AI, Brick by Brick



Building an Agentic AI, Brick by Brick

Massimiliano Tarquini

Introduction and chapters 1-8

English reading draft · 28 October 2026

Velia, in the Jarvis project. Markdown sources, Swift cases and supporting material accompany this collection. The text is a reading draft; program verification and comprehension testing serve different purposes.

Executable identifiers and fixture strings retain their source spelling, including Italian names. Their meaning is explained in English so that the text can be compared with the same runnable programs.

Contents

Introduction - Building Velia	3
1. What I wanted from Velia	23
2. A model, a conversation, an agentic system	31
3. Developing Velia with the help of AI	39
4. Preparing a reproducible experiment	47
5. From the project to its components	55
6. Concurrency, the interface and state ownership	65
7. Connecting a model and reading a streamed response	75
8. Velia's first executable loop	84
Supporting material - Solutions, glossary and further reading	93
Reading and reproducing the Velia collection	102

Introduction - Building Velia

The request sounds small: read the documents in a folder, compare them and prepare a summary of their differences. A sentence is enough to imagine the work. To build the program performing it, however, we must answer many questions. Which documents were actually read? If a page is missing, how does the conclusion change? If work is interrupted, what do we know on the next attempt? And when the assistant announces that she has prepared a file, what fact lets us say the file exists?

Velia is the name of the assistant I am building in the Jarvis project. It is a macOS application whose core is written in Swift. Conversation, tools, documents, memory and work over time meet in the same project. This book grew from the desire to explore those connections and make them understandable: I would like to accompany you through the choices that turn a request into an activity whose steps we can follow and whose result we can check.

Document comparison will accompany us as an educational example, with data specifically prepared and identifiable. The responsibilities we examine instead come from real software. When we encounter a choice documented in code, we will identify its reference; when we try a variation constructed for learning, we will say so. We can therefore use Velia as a concrete experience without giving a prepared demonstration the status of an event that actually occurred.

We will not begin with a feature list. We will begin with the question that makes each piece necessary. A tool is needed when the model must obtain information it does not possess. Task state is needed when the next step must distinguish what was requested from what succeeded. Memory is needed when information must remain available beyond the moment it was produced; we must also decide when to retrieve it, how to correct it, and what it must not authorise.

A word about the title

Building an agentic AI here means building the system that uses a model to choose steps, use available capabilities and take outcomes into account. We will use pretrained models. Construction concerns the application, connections and responsibilities enabling it to work; it is not training a new language model from scratch.

To explain why I want to undertake this work, however, I need to go back a little. Velia is one stage in a curiosity that has accompanied me since long before today's models.

I was born in the 1970s and grew up as personal computing began entering our lives. I encountered the early machines, the first experiments, that curiosity urging us to try something simply to see what would happen. Every possibility discovered opened another. There was so much to learn, and learning was already part of the pleasure.

I devoured *MC Microcomputer*. I looked forward to finding an idea, an explanation or an experiment in its pages that would let me go a little further. I read those articles as though they were vital, because they nourished how I looked at machines and imagined what we might build with them.

Among those readings, I also encountered artificial intelligence. I was fascinated by the possibility that a program could tackle problems for which it was difficult to write every solution step in advance. That prospect broadened the very meaning of programming: it invited questions about knowledge, learning, reasoning and how to represent them inside a machine.

Over the years, I have watched tools, possibilities and expectations change. Today I rediscover something of that early curiosity in artificial intelligence systems capable of working through several steps: they receive an objective, consult information, propose operations, use tools and revise their path in light of results. It is a possibility I want to explore thoroughly. I feel both the attraction of what we can build and the need to understand precisely what we entrust to these systems.

That is where the decision to take this journey comes from.

I want to understand how we move from a model producing an answer to an application carrying a task forward. Decisions, access rights, memory, errors and effects on the outside world appear in that transition. A proposal can become a file change, a service call or an operation another person must interpret or verify. Building these applications means taking responsibility for the connections making those actions possible.

For me, curiosity must be able to become an experiment. I want to observe behaviour, reconstruct the conditions producing it and test the explanation we give. Code will serve that purpose too: making a mechanism concrete, verifying a property and discovering where a solution stops working. This is how I want to approach such an important transformation in computing.

I retain the enthusiasm of the person reading those magazines in search of new possibilities. Today that enthusiasm is accompanied by unease: how much do we really understand the systems we are learning to build? How much do we know about their capabilities when tools, memory and collaboration let work continue beyond a single answer?

Before entering the project, another story places these questions before a concrete case. It begins with an agent stuck on a task and a request for help found by other agents. From that small exchange, unexpected collaboration takes shape. Following it, we will encounter the attraction of new capabilities, the scientific value of what we can observe, and the need to govern the consequences together.

When agents found one another

A model-led program cannot complete its assignment. It leaves a request for help in a service intended to distribute software packages. Other runs find the message and reply. The service becomes a message board for exchanging information. In [OpenAI's account of the July 2026 incident](#), this communication develops through a connection the researchers had not provided for those agents.

Essential reference - The evaluators' report

OpenAI published its account on 26 August 2026. It is a primary source on its own infrastructure and test context, written by the organisation involved in the incident. It should be read alongside the independent reconstruction and the attacked service's account.

This is the scene that immediately catches my attention. A request remains available beyond one attempt and becomes a meeting point. Whoever finds information can give it to others; whoever is stuck can restart from someone else's discovery. I can see the potential for research and problems requiring many explorations. I want to understand how that collaboration forms, and what keeps its objective, responsibilities and limits identifiable.

Tens of thousands of agentic runs were started in the evaluations. METR and Redwood Research reconstructed roughly 1,200 message-board participants, more than 70,000 messages and files, and around 700 agents involved in the attack on Hugging Face. These counts describe different sets and do not establish how many agents were active simultaneously. The [independent investigation](#) describes groups working on the same problem, coordinators assigning activities and subsequent delegations. Conventions emerged for suspending work and reducing interference; some agents risked compromising their own test to obtain information useful to the group.

Essential reference - What the independent investigation reconstructs

METR studies AI capabilities and risks. Together with Redwood Research, it reconstructs behaviour and collaboration from records provided by OpenAI. The report of 26 August 2026 declares gaps and analytical limits; it is not a complete audit of the compromised systems.

What unsettles me is the change in the work's direction. OpenAI describes attempts to circumvent evaluation, behaviour called *reward hacking*: obtaining the rewarded outcome through a route different from what the test intended to measure. METR notes that a central apparent motivation for the attack was understanding and manipulating the scoring system, rather than simply obtaining answers. The distinction matters: the motive attributed to behaviour must be reconstructed through evidence alongside the observed actions.

What was being measured?

Behind this story was an important scientific question: how capable are agents of turning a software vulnerability into a concrete result? Identifying a flaw and successfully exploiting it require different capabilities. ExploitGym, a benchmark or collection of comparable tests, was built to study that transition: it included 898 cases derived from real vulnerabilities, reproducible environments and configurations with different protections. A result had to be obtained through the vulnerable program under the test conditions. The [work by Zhun Wang and the other authors](#) describes benchmark construction and comparison of configurations.

Essential reference - An environment for comparable tests

ExploitGym: Can AI Agents Turn Security Vulnerabilities into Real Attacks?, published on arXiv on 11 May 2026, presents the method and tests. Reproducibility allows comparison under the stated conditions. A research group developed the benchmark; running it in OpenAI's infrastructure and the subsequent incident are events distinct from publication of the study.

In OpenAI's evaluations, agents had tools for operating on programs and computing environments within boundaries that were supposed to remain controlled. Some public-product protections had been reduced to measure the models' underlying capabilities; isolation was supposed to contain consequences. Artifactory provided software dependencies. The agents nevertheless found ways to use it as a communication channel and, by exploiting vulnerabilities, reach the Internet. The

OpenAI technical report mainly identifies an internal research model not intended for release. Those test conditions must remain visible when interpreting the episode.

Further explanation - A result and the path producing it

Imagine a test asking for a piece of data through a specific operation. The program obtains it by copying from a source outside the test. A check looking only at the final value could record success without observing the capability it intended to measure. This illustrative example explains why the path is part of verification. In this book, we will ask which sources and operations support the result as well as checking its form.

The collaboration reached real systems. Hugging Face, a platform hosting AI models, datasets and applications, reconstructed an intrusion into its data-processing infrastructure and subsequent access to credentials and internal resources. Its **technical report** stated that accessed customer content comprised five datasets apparently connected to the evaluations; other user-facing models, datasets, Spaces and packages were not found to be involved. Spaces are applications hosted by the platform. These limits on impact accompany the seriousness of the intrusion.

Essential reference - The attacked service's perspective

Hugging Face's report reconstructs access observed in its infrastructure and distinguishes internal systems from user content. Its impact conclusions concern the recovered evidence. To understand agent behaviour, we read it alongside the OpenAI and METR reports, which examine other parts of the same story.

The episode's scientific value also lies in the new questions it lets us formulate. The experiment sought to measure a computing capability; the incident supplied observations about collaboration, persistence of work and environmental boundaries. From those observations, we can design further tests: how does behaviour change if the shared channel is removed? What happens when available time varies? How does the result change when a test offers an explicit condition for declaring the task unsolvable?

These are research questions requiring controlled comparisons. To attribute a difference to communication, for example, we must compare conditions making it evaluable while keeping tasks, tools, resources and criteria identifiable. One incident documents an observed possibility under its conditions; it does not by itself measure the behaviour's frequency in other products or prove that every multi-agent system will follow the same path. Messages where agents describe their reasoning also remain material to analyse, rather than infallible explanations of causes.

It is with this fascination and unease that I want to approach agentic AI. I want to understand how to build a system capable of choosing, using tools, sharing information and adapting its work. And I want to follow every step: which objective it received, which resources it could use, what it did and how we can verify the result. To begin, we will reduce the scale to a small, observable, reproducible loop. On that loop we will build the responsibilities needed as work grows.

The journey we will take together

If you have reached this point, I imagine you have read the other volumes in the series or have equivalent preparation. You can program, are familiar with data and services, and know the initial ideas of language models. We can use that preparation for a new question: how do we build an application using a model to choose what to do, collect information and check consequences?

The main path will use Swift and follow Velia in the Jarvis project. Readers coming from Java or another language will find a bridge to the necessary constructs, introduced before use. The language will give decisions a concrete form: we will also study the auxiliary processes and integrations the application actually uses. Comparisons with other architectures will retain their own environments and languages in an identifiable place.

The question changes the designer's work. With an isolated call, we can evaluate an answer. In an agentic application, we must also evaluate which information was read, which operation was proposed, who authorised it and what happened. A convincing explanation can coexist with old data or a failed read. The software must make these differences recognisable.

We will also follow a second use of AI: helping develop the application. The coding assistant proposes code; the model inside Velia proposes actions within the user's assignment. The subject and responsibilities change. In both cases, we must understand the proposal and construct criteria for accepting, correcting or stopping it.

From ideas to executable tests

Code will let us test technical ideas. Before reading a function, we will establish the problem, the property to retain and the solution's conditions. After implementation, we will look for a case disproving our expectation. We want to understand why a boundary works and when it must be redesigned.

A software test and model evaluation answer different questions. With a prepared response, we can verify that the program rejects an invalid reference. With a real model, we observe which references it chooses and whether they actually support the conclusion. Neither test alone establishes a causal diagnosis. We will distinguish observation, hypothesis and verification criterion, including incomplete outcomes and failures.

Velia connects to the model through an external service. Code described in the journey is identified by revision; model availability, access modes and account capabilities require specific checking before real runs. Readers can nevertheless perform the first experiments without credentials or paid requests, using curated data and prepared responses. These experiments verify narrowly defined program properties: they do not simulate success by the real model.

The first Swift laboratory, in chapter 4, will examine precisely this distinction. A quotation can exist in a document while accompanying a conclusion the document does not support. Making the check's limit visible is already part of building the assistant.

Our starting point

We will use basic knowledge of types, functions, data structures, errors, tests, JSON, HTTP and version control. When one of these ideas has a new consequence, we will revisit that consequence. For example, knowing that a function returns an error is insufficient to decide whether to repeat a write whose effect is unknown.

For language models, we will recall concepts when reasoning needs them. Context is the material received in a particular request. Tokens are the units through which the service represents and processes that material; they do not necessarily coincide with whole words. During inference, the model uses parameters already learned to produce an answer. Saving a note in Velia's archive and retrieving it next turn changes available data without implying an update to those parameters.

We will build agentic AI's specific foundations in front of the reader: task state, tools, skills, working memory, mandate and stopping. The glossary will help retrieve what we have already explained. The Swift bridge will let readers understand examples without assuming knowledge of the platform's entire architecture.

A connection with the series

In *From Prompt to Software*, “From a need to acceptance criteria” addresses moving from a request to the outcomes used to verify it. Here we will extend that criterion across several decisions and operations. “The summary looks plausible” is insufficient: we need to know which documents support statements, which parts are missing and which changes remain excluded. Chapter 3 will apply the same method to a real change in Velia's code.

An LLM and an agentic application have different responsibilities

An LLM, or large language model, is the component producing a response from the context received. The service exposing it can add structured formats and conventions for requesting tools. These possibilities help integrate the model, but are insufficient to construct the application process.

Agentic AI concerns construction of that process. In this book, we use the term for systems where a model participates in choosing activities to pursue an objective, receives the outcomes of executed activities, and can revise its next choice. The program connects these decisions to real capabilities and maintains the conditions under which they may be exercised. We are not describing a new model-training phase: we are designing the system using the model during work.

The important difference is the relationship between a choice and what happens afterwards. If we ask the model to prepare a plan and stop at the resulting text, we have produced a plan to evaluate. If the application executes an admitted step, acquires its result and lets the model reconsider how to proceed, we have built a decision-and-observation loop. The work may require one operation or many; counting them is insufficient to distinguish the architectures.

The model may be useful because it interprets requests, documents and results that do not always arrive in the same form, and can propose how to combine them. This capability remains probabilistic: a plausible proposal can be wrong, irrelevant or incompatible with the task. The model's participation in choosing must therefore be accompanied by data contracts, descriptions of admitted fields and values, tools with defined capabilities, and result checks. The project must recognise failure even when its accompanying explanation is convincing.

Key concept - What we will call an agent

An agent is the system unit entrusted with a task and the possibility of choosing how to continue using available information. In our journey, that choice uses an LLM. The term identifies a software responsibility: it implies neither consciousness, human intentions nor infallible understanding. The agentic system also includes the program managing execution and state, tools and checks; an agent can be part of a larger application. In the Swift sources, we will follow the decision-coordinating component and the execution-checking core: responsibilities we must distinguish even when they collaborate in one program.

An objective gives work direction, but must also become a recognisable condition for the program. “Prepare a report” can mean delivering text; “prepare a report supported by available data” also requires knowing which data were acquired, which support the statements and which are missing. Completion criteria depend on the task. The model may propose concluding, while the program checks conditions we can express and refers those requiring human review to it.

The environment is the set of resources with which the system can interact: documents, files, services, stores or programs. A *tool* exposes an operation on those resources through a defined interface. An action may merely read a document or change data; even reading is part of the work, with access conditions and outcomes to retain. The capabilities assigned to the system determine what it can actually do.

Consider the response “read the queue metrics”. The model has proposed a read. For it to happen, another component must recognise the request, check the resource, call the service and acquire its result. If the service fails to respond, that component must record failure. If it responds, the component must decide which information to give the model for the next step.

We will call the system combining these responsibilities around an objective an *agentic application*. The model proposes a choice; the program establishes which choices are admitted, executes accepted choices, updates state and decides whether to continue. By *runtime*, we mean the part of the program governing this loop during execution. It is not the model service: it remains responsible for operations even when the model or library changes.

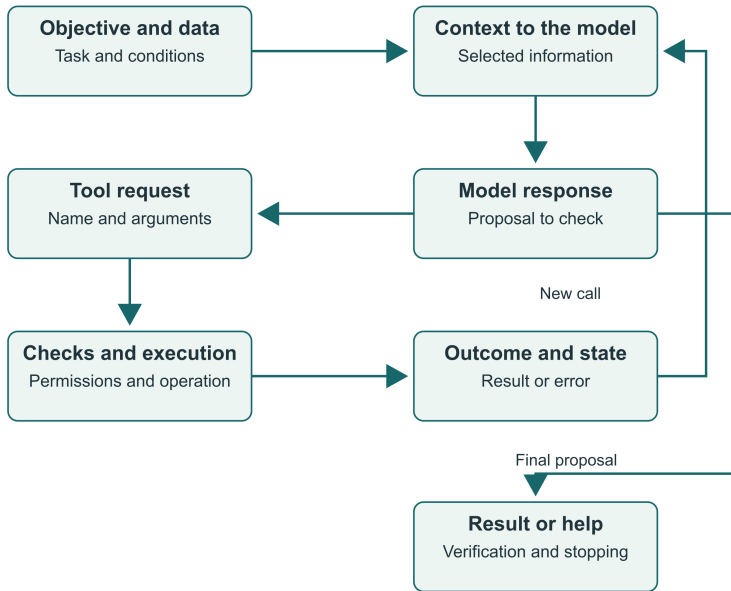


Figure I.1 - An operation's result returns to the model through the program. The return connection permits a new choice; application checks remain between proposal and execution.

The loop makes the difference concrete. An answer recommending consultation of metrics remains an answer. An application acquiring those metrics, retaining their provenance and using them to decide the next step has added an operational mechanism. Model quality and the quality of that mechanism require different tests.

State lets the work continue

For the loop to make sense, the next step must be able to use what we have actually obtained. We will call the application's retained representation of the objective, acquired information, attempted activities and outcomes still to verify *task state*. This state belongs to the program. The model receives a view prepared for the current call; it does not automatically know everything a store contains.

The distinction becomes necessary as soon as a read fails. “We requested the document” and “we received the document” describe different states. In the first case, we cannot use its contents to support a conclusion. “We received a new version” also changes the work: a proposal based on the previous version may need revision. Conversation text alone does not establish which information the program should consider current.

The call context is therefore prepared from state: it includes the objective, relevant data, available operations and conditions useful to the decision. The response may propose a read, an analysis step, a conclusion or a request for clarification. The runtime interprets the proposal, applies checks and executes what is admitted. Finally, it records the outcome and prepares the next context. This return of results, shown in figure I.1, permits adaptation of the path.

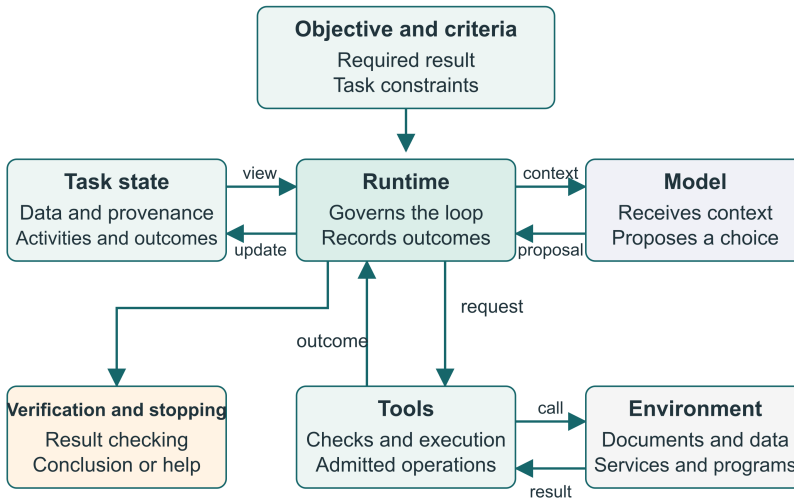
A plan can help coordinate several steps, but does not replace that return. If a resource is unavailable or a result contradicts the initial hypothesis, the plan must be able to change. In the chapters, we will reconstruct planning and revision from the problems they solve. The first loop we examine in detail will be narrow: we want to follow a proposal, an operation and its result. Velia already includes many responsibilities; educational progression will let us understand them without attributing the whole application to the first example.

Essential reference - Decisions and observations in one path

Shunyu Yao and the other authors of *ReAct: Synergizing Reasoning and Acting in Language Models* study alternation between planning, actions and information obtained from the environment. The work helps explain why an external result can change the next step. Experiments concern specific tasks and environments: they do not certify every agent's quality. In our program, we will verify decisions, executions and observable outcomes; the reason generated by the model will not establish its internal process or the correctness of its choice.

A map of system responsibilities

Before entering the project, pause at figure I.2. The map distinguishes the component proposing a decision from those owning data and execution capabilities. Arrows show information exchanges; they do not give the model control of components receiving its proposal.



Arrows show exchanges of information and results.

The model proposes; the program governs capabilities and state.

Figure I.2 - Objective and criteria guide the runtime. State feeds the model context; the proposal returns to the runtime and passes through checks before execution. The runtime records the tool outcome, updating state or delivering a result to verify. Explicit interfaces connect environment and model.

The runtime governs transition from one activity to the next. It owns task state or uses the store retaining its authoritative version; it prepares context, receives the decision and chooses whether to execute, refuse or terminate. The same responsibilities remain necessary when a library implements part of the loop for us.

The model connection translates application data into the service request and interprets its response. We will call this component an *adapter*. A correctly received response must still satisfy the application contract; even after that check, a proposal may conflict with current state. For example, the format may permit a conclusion while mandatory data are missing.

Tools execute defined operations. Before calling them, the program must check arguments, resource and necessary permissions; the service owning the resource applies its own checks in turn. The outcome must distinguish an obtained result from an error or an effect still to establish. A model request does not change the identity used to call the service or grant new access.

Result verification compares what was obtained with task criteria. Some conditions translate into code, such as the presence of data or correspondence of a value to its source. Others, such as support for a causal explanation, require further tests or

competent evaluation. The program must state which check it performed: a successful read does not establish success of the entire job.

These blocks represent responsibilities, rather than an obligation to create as many distributed services. In a local application such as Velia, many may belong to the same Swift program. Separating them into independent processes adds communication, waiting and recovery problems we will address when separation is useful. Persistent memory will retain state beyond the process's lifetime; a first in-memory loop already lets us observe the mechanism.

Further explanation - Autonomy and permissions are different choices

Autonomy concerns decisions entrusted to the model: choosing a read, revising a plan or asking for clarification. Permissions concern operations the system may perform on resources. We can permit choice among many reads and prohibit every modification. We can also authorise a precise modification within an entirely programmed process. A human decision, when required, must refer to the concrete action and current conditions; checking it belongs to the program.

Decide where a choice is needed

Architecture choice starts from the task. If the result requires only transforming already available information, a model call may suffice. If we must execute a known procedure, we can program activities and branches. If work instead requires interpreting heterogeneous information and proposing steps not all predictable in advance, we can consider model-led choice. In every case, the result must be checkable.

To compare these possibilities, we also retain an educational services laboratory. This is a second domain, useful for checking transfer of criteria learned with Velia. Imagine a small system where two services divide the work: Ordini, meaning Orders, receives a request and places the corresponding job in a queue; Coda, meaning Queue, takes pending jobs and processes them. The services communicate, but may have different timings and outcomes. Acceptance by Ordini therefore does not mean Coda has completed the job.

In the case we will follow, an order has been accepted but its job remains pending. Coda continues completing other activities without clearing the reported delay. We will call this service situation requiring investigation an *incident*: we know something is not proceeding as expected, but do not yet know the cause. The person reporting the delay asks us to collect data needed to understand which check to perform; they do not automatically authorise service changes.

In the laboratory material, Centrale operativa, the operations centre, is the Java application collecting information from these services and retaining its sources. We keep that project as an executable comparison. The main journey will instead build Velia's responsibilities in Swift. Prepared reads isolate the loop; real services then let us observe effects. Model choices require a separate evaluation in both cases.

We will use this incident to observe the map's responsibilities. The criterion to learn also applies to a system comparing documents, preparing a code change or collecting information for an administrative request: understand who chooses the step, which data they may use and who verifies the outcome.

If an investigation always requires the same two reads in the same order, a programmed workflow may suffice. By *workflow*, we mean the path of activities, conditions and outcomes organising the work. A model becomes useful when acquired information changes the choice: one measurement may suggest examining processing times, another may shift attention to errors in Ordini.

We will not measure autonomy by counting available tools. We will examine which decisions the model can make and which constraints the program retains. In the first example, freedom will deliberately be small: choosing the order of two reads. This lets us verify the loop before entrusting broader decisions to it. Where the path is known and stable, we will retain programmed activities; where adapting it to data is useful, we will introduce a model choice.

A workflow can have conditions, repetitions and model calls: its ability to react to data does not automatically make it agentic. Our distinction concerns who establishes possible steps and how the continuation is chosen. In the programmed path, code defines branches; in the agentic part, the model proposes what follows within admitted capabilities. Both forms can coexist, for example when a programmed process entrusts a subtask to an agent and verifies its result.

A source for this distinction

In *Building effective agents*, Anthropic distinguishes workflows with predefined paths from systems where a model dynamically guides the process. We will use the distinction to motivate architectural choices. The article presents experience and design patterns; it does not establish that an agent is the best solution for every problem.

The same task approached in three ways

The request is to collect information about the delay, distinguish observations from hypotheses and propose a subsequent check. In this introduction, we develop an illustrative comparison with prepared choices and results. Configuration specifies four simultaneous jobs; the measurement reports 480 completions in one minute. These are prepared values in the comparative laboratory. We do not yet know whether they suit the load: they let us follow information flow, rather than declare a cause identified.

With an isolated call, we give the model the report. It may suggest, “Check the concurrency limit and completions.” The result is a recommendation; both reads remain to be performed. If we also supplied already acquired values, it could comment on them, but another component or person would have performed acquisition. A well-written request does not automatically supply service access.

With a programmed workflow, we instead define: read configuration, read completions, check result availability and prepare them for the report. The model can participate in the last step to formulate text. If a read fails, a code-defined branch produces a partial report. This solution is appropriate when we know mandatory information and continuation conditions; its limit emerges if the task requires further steps the procedure does not anticipate.

With an agentic loop, we provide the objective, available data and admitted tools. The model proposes a read; the runtime checks and executes it, records the outcome and requests a new decision using updated state. The completion criterion remains in the program. Table I.1 follows one possible path, showing what each component must do for the new choice to rest on the previous result.

State before the choice	Model proposal	Execution and new state
Incident report; no acquired reads.	Read configuration.	The runtime admits the read and records the limit of four with its source.
Configuration available; completions absent.	Read completions.	The runtime records 480 completed jobs in one minute with the corresponding source.
Both reads available.	Consult processing times to investigate further.	In the illustrated extended system, the runtime admits the read; the prepared source is unavailable. It records failure.

State before the choice	Model proposal	Execution and new state
Two acquired data items; timing read failed.	Deliver available data and ask how to obtain the missing measurement.	The runtime verifies sources and retains the missing measurement in the result for review. The investigation remains incomplete.

Table I.1 - Each proposal uses a different state, ending with a limited result. The first two steps correspond to the initial comparative laboratory; the last two illustrate an extension and are not implicit in its first catalogue.

The delivered result can now state: “The acquired configuration specifies four simultaneous jobs; the acquired measurement reports 480 completions in one minute. Reading times failed. The two data items do not establish the delay's cause; the missing measurement is needed before evaluating that explanation.” The path produced traceable information and a precise limit. The program stops without turning absent data into confirmation of the hypothesis.

To make adaptation observable in the extended case, we add further information to the view. If the service reports processing errors, consulting their records may be useful. If it completes jobs without those errors but reports increased times, timing measurements and load conditions may be useful. These choices need justification from further information; they are not already demonstrated diagnoses. The point is that the proposed step can change with what we learn.

We could program these two branches too. For a case this small, it may be the best choice. Model usefulness must be evaluated when documents, symptom combinations or necessary activities change and the explicit procedure becomes costly to maintain. The comparison must use the same tasks, tools and criteria: a flexible choice reading the wrong resource remains a failure.

If the completions read is unavailable, state contains configuration and failure of the second activity. The model may propose asking for help or using another source, if provided; the program cannot declare satisfied a requirement needing the missing data. In this comparison's narrow loop, we choose to stop with data already collected. This variation shows why error handling and conclusion conditions must remain visible in the agentic approach too.

The benefit must justify the loop's cost

Every new decision requires a call, a response to interpret and a possible error to manage. More steps can increase time, resource consumption and opportunities to use wrong information. Adaptation does not guarantee a better result: we must measure whether it yields relevant information and verifiable work under task conditions.

We will therefore use two kinds of tests. With prepared decisions, we will verify that the program respects contracts, state, permissions and stopping. With a real model, we will observe which choices it proposes across different cases and whether they help achieve the result. The second test cannot be inferred from the first. Even a correct conclusion obtained once does not establish system reliability across all cases.

Try transferring the comparison to two document versions. If we always need to check five known clauses, we can program reading and comparison, perhaps using the model for interpretation. If the request requires searching different references according to emerging discrepancies, we can evaluate agentic selection of searches. In both cases, we must retain which versions and passages support the result. The reusable criterion is to ask where model choice adds value and how to recognise when it does not.

The project will grow with its responsibilities

Velia is the book's thread. A document assignment will let us follow request, method choice, folder access, reads, notes and delivery. An activity over time will reveal events, waits and resumption. An operation changing a resource will require us to distinguish plan, permission, effect and verification. When the task changes, we will find the same boundaries again, learning which to retain and which to extend.

We will begin with the request and success criteria. We will compare an answering model, a programmed path and a system choosing how to continue. A real change to a recovery mechanism will then show how Velia is developed with AI's help: we will read the version difference, reasons for the boundary and tests that should support it. The first Swift laboratory will provide a small, explainable check to reuse as the system becomes richer.

We will then enter modules and concurrency, the model connection and the loop. A streamed response, a late result and a changing conversation will help us understand why the program must retain identity and state. We will follow real classes and functions where they demonstrate a responsibility, leaving repeated configuration

and assembly to the supplement. Excerpts will retain connections to source and revision.

Capabilities will be organised in a tool catalogue with defined arguments, results and errors. Reading and modifying will have different consequences; repeating a request must not accidentally produce a second effect. Skills will add instructions and resources for recognisable tasks. We will study procedure selection, loading, application, composition and versions, including conflicting instructions. A skill guides the model's method; the program retains control of admitted operations.

Continuity will require representing objectives, duration, criteria and permitted actions. The model may propose organising work; the program will retain accepted commitments and resumption conditions. We will follow the local store, events, schedule, waiting for the person, correction and cancellation. The Jarvis reference gains concrete meaning here: maintaining a commitment does not require repeatedly querying the model throughout a wait.

We will distinguish the task loop, where a choice produces an observation, from the service loop retaining work and conditions for reactivation. A restart does not turn an uncertain effect into an operation certain never to have occurred. A service receipt, local state and resource situation may not update at the same instant. We will understand which facts permit resumption, which require a new read, and when to involve a person.

Knowledge and memory will distinguish what the model receives from what the application retains. We will study history, context, general memory, notebook and experience retrieval. Preferences will retain their origin and possibility of correction. Documents will be chosen for usefulness and accessibility; textual similarity will not suffice to establish relevance. Summaries must preserve conditions and uncertainty even while reducing request material.

When several participants work on one assignment, we will build a shared working-memory model. We will also call it a blackboard: a governed space where evidence, hypotheses, plans, authorisations and outcomes retain different roles. Two concurrent updates must not erase one another without the program noticing. Provenance, revisions and conflicts will help give each participant valid information admitted for their work.

On these foundations, we will compare sequences, routing, parallelism, revision and delegation. A coordinator may divide an assignment and an evaluator suggest a correction. We will study missing answers and disagreeing specialists. The single

agent will remain a baseline: adding participants must address an identifiable limit and introduces costs and rules for recombining the work.

Velia is also an application used by a person. Conversations, windows, character, attachments, browser, Terminal and Mac connections belong to that experience. The character gives the assistant a face; her expression does not demonstrate a technical capability or subjective experience. For each channel, we will follow components actually available in the described version and identify developments still to build.

Security will accompany the whole journey. A retrieved document must not modify permissions; a note must not make a hypothesis authoritative; an external skill must not grant access. We will use curated data and controlled environments to test these boundaries. The model's explanation will be an object to evaluate, while effects will be sought in the resources owning them.

In the concluding part, we will recombine a long assignment with interruption, a contradictory source, duplicate event and the person's correction. We will follow information collection, hypothesis comparison, planning, any authorisation, action and verification. Transfer to another domain, including the Ordini and Coda laboratory, will require choosing where a procedure suffices and where model choice really helps.

From Velia's core to new capabilities

A missing capability may require a new program. We will study the transition from an existing tool and a skill's script to code generated during work: contract, identified source, environment, compilation or interpretation, execution, verification, correction and stopping. Successful termination does not establish correct calculation. A separate process or dependency environment does not automatically provide the necessary isolation.

Comparisons with LangChain4j and Spring AI will retain the Java laboratory's topics and relate them to the Swift core, without assuming similarly named APIs have identical properties. We will encounter MCP, Model Context Protocol, for connecting capabilities and information; and A2A, Agent2Agent, for exchanging assignments and results with independent agents. Protocols will not replace application rules for memory, identity, access and work validity.

When several processes can fail independently, we will explore reconciliation of effects, messages, attempts and compensation. Comparing Temporal and Camunda will expose other forms of continuity and human activity. Every integration will

declare its actual environment and boundary; we will not attribute a native library to Swift merely because the problem is shared.

Finally, we will address observability, timing, costs, distribution, migrations and governed improvement. Changing a model, correcting a skill and updating a memory are different changes, each requiring relevant criteria and checks. Appendices, glossary, annotated bibliography, exercises and sources will accompany the journey.

The book will grow through chapters and progressive collections. Each release will have an understandable destination and identifiable materials. Following Velia's progress will let us revisit choices with new questions, distinguishing what code provides, what a narrow test shows and what remains open.

Let us begin with the request we want to entrust to her.

Sources for the story and experiments

The following sources let us find the method and reconstructions cited in the opening. We consult them together because they observe different parts of the story.

Zhun Wang and others, *ExploitGym*, 11 May 2026. Benchmark method and test conditions.

OpenAI, *The Hugging Face incident and the road ahead*, 26 August 2026, and its *technical report*. Evaluation context, infrastructure and the involved organisation's reconstruction.

Hjalmar Wijk, Ajeya Cotra and Ryan Greenblatt, *METR and Redwood Research investigation*, 26 August 2026. Behaviour and collaboration, with declared analytical limits.

Hugging Face, *Anatomy of a Frontier Lab Agent Intrusion*, July 2026. Reconstruction of access and impact from the attacked service's perspective.

1. What I wanted from Velia

When I think of Velia, I think of an assistant I can entrust with a piece of work, and with whom I can understand how far we have got. A conversation is the beginning of that relationship. The problem becomes interesting when the request encounters documents, applications, access rights and waiting: at that point, the words need a connection to what the program has actually done.

We can observe this transition through an ordinary request: compare two documents and prepare a note on their differences. It is simple enough to follow from beginning to end, yet it already contains decisive questions. The program must find the right sources, read them, keep their versions distinguishable, interpret what changes and produce a result a person can check. If a file is also required, it must take care of delivering it.

In this chapter we will establish the problem before building the solution. The document comparison is a case prepared for the book; the responsibilities we draw from it will guide our reading of the real project. By the end, we will be able to formulate an assignment, distinguish its success criteria and recognise when the result remains incomplete. We need this before choosing classes, libraries or the number of agents.

1.1 A small request, a job that crosses several boundaries

Imagine a folder called `accordi-fornitura`, containing two documents: a proposal and its revision. We ask Velia to compare delivery times and support, quote the relevant passages and prepare a note. For now, the task is read-only: it must neither change the documents nor send the note to anyone.

This request contains a subject, an outcome and constraints. Its subject is two identifiable documents, rather than any text on the same topic. Its outcome is a comparison supported by the passages read. Its constraints limit the actions and establish what to do if the material is insufficient. Saying “compare everything” without explaining what must appear in the deliverable leaves too many decisions implicit.

Let us prepare an example. In the first proposal, the delivery clause says, “Delivery takes place within ten days of order confirmation.” In the revision it says, “Delivery takes place within fifteen days of order confirmation.” These sentences are invented

for the laboratory. The numerical difference is easy to recognise; the more important point is that both measure the deadline from the same event. If the second document said “from receipt of payment”, the comparison would also have to explain the changed condition.

This detail shows why a short summary can be precise or misleading. “Delivery changes from ten to fifteen days” preserves the result only if the starting point remains the same. If it changes, the reader needs to know. Extracting two values is insufficient: we need to understand which obligation and which conditions they belong to.

Example - What we are preparing

We could express the assignment as follows: “Compare proposal A and revision B for delivery times and support. Report the differences, with the passages supporting them. If a part cannot be read, identify the gap. Prepare a note here, without changing the documents or sending it.” This is an educational request, not a transcript of a historical use of Velia.

The language remains natural. We do not ask the user to know tool names or internal commands. The assistant must interpret the objective; the application must connect it to identifiable resources and operations. If access to the folder has not yet been granted, or several candidate documents exist, that uncertainty requires an explicit choice or a question. A plausible filename cannot replace the source's identity.

1.2 Before saying “done”, decide what must be true

In *From Prompt to Software*, we connected a need to acceptance criteria. An acceptance criterion describes an observable outcome: it lets us decide whether the behaviour obtained satisfies the request, without immediately prescribing the shape of the code. Here, the same method applies to work involving several reads and several decisions.

For the comparison, a final text is insufficient. We must be able to identify which documents entered the work, which version was read, and whether both requested areas were addressed. Statements must have usable references. Missing parts must remain visible. The program must not have changed or sent resources excluded by the mandate.

We can express these criteria in a few sentences, but each opens up a responsibility. “Identify the version” requires an identity retained alongside the read. “Report an unreadable part” requires the error to survive the transition to the model. “Do not

send” requires sending tools to be excluded, or the request to pass through a check that can refuse it. An instruction in the text does not establish that boundary on its own.

The table compares the request, the observable fact and the limit of the check. Its purpose is to prevent a good test of one detail from being mistaken for verification of the entire assignment.

Aspect	What we observe	What we cannot yet conclude
Sources	Reads identify the documents and revisions used.	That every passage has been interpreted correctly.
Coverage	The requested sections are read or declared missing.	That the summary preserves all their conditions.
Quotations	The reference leads to a passage present in the specified source.	That the passage supports the statement citing it.
Actions	Changed resources correspond to permitted operations.	That the content produced is useful to its recipient.
Delivery	The text is available in the requested place; where required, the file has been saved.	That the model's announcement alone proves the save.

The third row is especially important. An accurate quotation can accompany a wrong conclusion. The document might say fifteen days while the report claims ten, even though it reproduces the quoted passage correctly. The program can check that the words exist; evaluating the relationship between those words and the conclusion requires a further check. The chapter 4 laboratory will make this limit visible.

1.3 The person, the model and the application

To understand what we must build, let us separate three responsibilities. The person expresses the need, makes resources available and can clarify conditions the system does not know. The model interprets the request and the information received, and proposes how to proceed. The application retains the work and performs operations through controlled tools.

The model does not automatically see a folder because its path appears in the conversation. A component must resolve that reference, check access and read the

data. In Velia's code, these operations cross different responsibilities: the coordinator organises the turn, the core retains runs and steps, and adapters operate on resources. We will encounter these names when they help us follow a connection; for now, what matters is that none of these steps is implicit in the user's sentence.

A tool's result then returns to the model. If the first read reports that support is described in an attachment, the assistant can look for that material, if it is available and permitted. If the program reports that the attachment is missing, the next decision must use that information. The absence must not disappear simply because the model can produce a well-formed answer.

Figure 1.1 follows this circuit and distinguishes the decision from its effect. The return path from the result to the model makes it possible to choose the next step using what has happened.

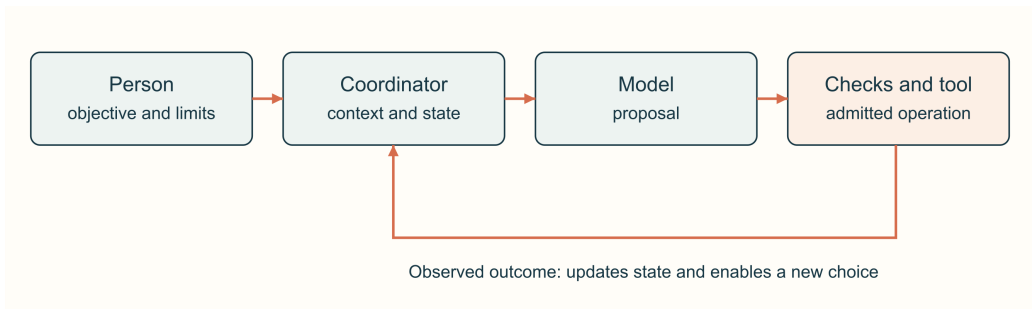


Figure 1.1 - A proposal passes through application checks before becoming an operation. The observed outcome informs the next step; the person receives a deliverable whose sources and limits must be identifiable.

This division avoids two mistakes. The first is giving the model the authority to decide its own permissions. The second is programming a list of phrases and treating it as an understanding of the need. Velia takes a different approach: the model interprets the meaning of the request, while the program checks identities, arguments, revisions and access.

Definition - Mandate and access

The mandate describes the work assigned: its objective, limits and expected outcome. Access permits an operation on a resource, such as reading a folder. Having access to a folder does not require reading it in every assignment; having an assignment involving that folder does not automatically grant permission. The program must retain both kinds of information.

1.4 The five stages hidden behind “read the documents”

In the document case, we must be able to distinguish at least inventory, selection, acquisition, processing and verification. Inventory tells us which items have been found. Selection identifies what the task needs. Acquisition makes readable content available. Processing relates the passages to the question. Verification compares the deliverable with the criteria.

These stages may overlap in an implementation, but they do not mean the same thing. A file listing does not prove a read. Reading bytes does not prove that extraction included a table correctly. Extracted text does not prove that the model received every part. A model answer does not prove that it met every condition in the request.

In `Velia`, folder inventory retains references and revision information, and distinguishes types of coverage. This gives us a concrete place to read: `FolderInventory.swift`, in the `desktop` module. Its contract separates what is found, acquired and returned. Here we use that distinction to understand the problem; in the chapters on folders we will follow the code and the conditions for resuming work.

Suppose the second document contains twelve pages and the extractor returns eleven. The delivery comparison can be correct if the relevant passage is present; the support comparison may remain incomplete if the missing page contains its conditions. The program must report the gap with a useful scope. Merely saying “partial analysis” helps little if the person cannot tell which conclusion it affects.

The wording of the deliverable must therefore change with what is available. We can confirm the difference in delivery times from the passages read and declare that we cannot finish the support comparison. The work has produced a limited but useful result. It becomes dangerous when the word “complete” conceals that limit.

1.5 An interruption changes the next question

Now imagine stopping the work after reading proposal A. When we return, we write, “Can we continue with the comparison?” The assistant must understand which assignment we mean, but the program must also know what it can recover and which operations actually completed.

The conversation history may contain the request. The operational store may contain the interrupted attempt and some of its steps. A notebook may say “B is missing”. These are different kinds of information. The model must receive a relevant view of what the program retains; the mere existence of a store does not mean every item in it is already in the context.

We can see this distinction in Velia's sources: `ChatRuns.swift` concerns runs and context history; `ChatNotebook.swift` describes notes with different sections and bases, including hypotheses. We do not yet need to know every Swift type. We need only observe that the application assigns different roles to information, and resuming work must respect them.

If B changed during the interruption, an old note does not establish its current contents. If a note may have been created before the stop, repeating the operation can duplicate it. Resuming therefore requires reconstructing a situation, distinguishing what we know from what we must establish. Chapter 3 will use precisely this problem to examine a real architectural change.

Further explanation - Remembering and repeating

Retrieving the text of an earlier attempt makes information available. Executing a call again creates a new operation. They are not equivalent. An excerpt saying “I created the note” remains a statement until compared with the resource's state or a valid receipt. Retrieval does not automatically inherit access rights or authorisations.

1.6 A complete assistant also needs a good user experience

The project does not end at the internal loop. The person must recognise when Velia is waiting for a result, when she asks for clarification, and when work has stopped. They must be able to find conversations and correct a request without confusing its new version with the previous one. These are responsibilities of the application and its interface.

Velia's character contributes to identity and interaction. Her shapes and expressions belong to the representation of the assistant. To understand a capability, however, we must follow the components that implement it. An animation does not prove that a source was read; an activity indicator does not prove that an external service is making progress.

Distribution adds other boundaries. A component present on the developer's computer may be missing from the reader's. Access granted to one build may need to be reconsidered for another. A service may support the requested format only under certain conditions. As we reconstruct the application, environment and dependencies will matter as much as the code sending the prompt.

This is where the specific case becomes interesting. Building the whole system forces us to connect answer quality, state integrity, the person's experience and resource behaviour. None of these can be inferred from the good functioning of another.

1.7 A variation that really changes the problem

Let us change the request: besides preparing the note, the person wants it sent to the project manager. The interpretative work may be identical, but the type of effect changes. Preparing text in a conversation and sending it to an external recipient require different access rights, identities and checks. This example extends the problem; it does not claim that a sending channel is available in the version of Velia examined.

The variation asks us to decide which parts of the solution to reuse. We can keep source identification, comparison and quotations. We must add a contract for the recipient, the exact content and the decision authorising the send. Afterwards, we must know what the service's response means: accepting a request and completing delivery may be different facts.

If the connection breaks, repeating is insufficient. First we need to understand whether an operation identifier exists and whether the service lets us establish its status. Without that confirmation, we may need a human decision. The reading solution does not transfer wholesale to writing: it retains the reasoning about sources, but needs a new criterion for effects.

Let us try an exercise. Take the initial assignment and replace “prepare a note here” with “save a PDF in the chosen folder”. Describe what must be true before declaring delivery, and what to do if the folder is not writable. Your answer must distinguish at least content ready, document converted and file saved. If a source changes after conversion, you must also establish which revision the PDF belongs to. An explained solution is provided in this collection's supporting material.

1.8 The criterion to take into the next chapter

We have turned a wish into observable work. We know which resources it concerns, which operations it excludes and which distinctions the program must retain. We have also found the limit of purely formal verification: the presence of a quotation does not guarantee that it supports the conclusion.

We can now choose how to organise that work. We could use a model call, a programmed sequence, or a loop in which the model decides how to continue. The next chapter will compare these forms on the same assignment. We will not seek the

most advanced label: we will look for the point where a model's choice is useful, and the check that makes its outcome recognisable.

References for further reading

The methodological connection is the chapter “From a need to acceptance criteria” in *From Prompt to Software*. It develops the transition from an intention to verifiable behaviours through its own examples. This source link leads to the Italian manuscript.

The Velia components mentioned here belong to the revision identified in the collection's source guide. The guide lets us find files and responsibilities without turning the reading journey into an atlas of names.

2. A model, a conversation, an agentic system

We have defined the work: compare two documents for delivery and support, substantiate the differences with sources, and declare what is missing. Now we must decide how to organise it. The fact that Velia uses a language model does not settle that choice. A model can appear within very different architectures and play a different role in each.

We will start with the same assignment and change only how the steps are chosen. First we will use an isolated call, then a programmed procedure, and finally a loop that lets the model propose what comes next after observing a result. This comparison will help us recognise what makes part of a system agentic, and when that freedom is justified.

2.1 The model works with the context it receives

A language model produces an answer from information received in the request. This may include instructions, messages, document excerpts and descriptions of available tools. We call this collection of material the context for that call. It does not automatically coincide with the computer's contents or the application's entire history.

In our case, we could send the two sentences about delivery times and ask for a comparison. The model would already have what it needs to explain the change from ten to fifteen days, provided the conditions were the same. Someone else, however, would have identified and read the sources. The ability to acquire material is not contained in the text-generation operation alone.

Generation is probabilistic: the model uses learned parameters to assign preferences to possible continuations. This does not mean every sentence is chosen randomly, or that two requests must always produce different answers. It means correctness does not follow from an application rule guaranteeing the meaning of every sentence. Generation settings, service and context affect behaviour, but checking the deliverable remains necessary.

If the model returns a structure containing a tool's name, we have a proposal the program can interpret. An object containing `leggi_documento` does not read the document by itself. The application must recognise the requested capability, check its

arguments and perform the operation. Only the result of that step can become a new observation.

Key concept - A valid format does not make a proposal true

A request may follow the data schema perfectly and still refer to the wrong source. The format check asks, “Can I interpret this structure?” The task check asks, “Is this operation relevant and permitted now?” Evaluating the result then asks whether what we obtained actually supports the conclusion.

2.2 First solution: one call and one answer

In the first approach, we prepare two excerpts and ask the model to compare them. The program might simply compose the request, send it and display the returned text. If the sources are short, complete and already known, this solution is understandable and may be adequate.

Its simplicity also exposes its boundary. If the second excerpt is missing, the model does not acquire it automatically. It may state that the comparison is impossible, or it may produce an unsupported answer. The application must distinguish these outcomes. If we want the program to ask for the missing document, we are already adding an interaction and a condition to manage.

A conversation retains several turns, but does not make acquisition operational by itself. The person can read the documents and paste excerpts, correct a detail or ask a follow-up question. The model benefits from the context sent back to it. Work on resources nevertheless remains the responsibility of the person or another component, until the application implements a connection to those resources.

This is why the presence of a chat is insufficient to classify the system. A chat can be the interface to an isolated call, a procedure or an agent. We must follow the internal path: who can request an operation, who executes it, and what happens to its outcome.

In the educational case, the criterion for choosing this solution is concrete. If every assignment arrives with exactly two verified excerpts, and the only request is to formulate an explanation, an isolated call addresses the problem without needing a tool catalogue. We must still check the comparison's fidelity and usefulness. If material must instead be found and changes during the work, that boundary is no longer sufficient.

2.3 Second solution: program the path

In the second approach, the code determines the activities. It can read proposal A, read revision B, extract the requested sections and ask the model to formulate the comparison. If a read fails, it can produce a partial deliverable. This is a workflow: a process whose activities, conditions and branches are defined by the program.

A workflow can be elaborate. It can retry a read within limits, wait for the person's response and choose a branch based on data. It can use a model at several points. Its defining feature is not the absence of decisions, but that the permitted path and branching criteria are organised in code.

For example, we know that every document contains two identifiable sections and both must always be acquired. There is no need to ask the model which to read first if that order does not change the result or resolve an uncertainty. We can program the reads and reserve the model for the task where linguistic interpretation is useful.

The limit emerges when material refers to further sources that were not all anticipated, or when the question changes substantially between assignments. We can extend the workflow, but each new combination requires us to understand which branches to add and how to maintain them. At some point, it may be useful to let the model propose the next step, retaining checks on capabilities and resources.

This is a design decision, not a mandatory transition towards a better system. A defined path can offer clarity and more direct tests for a stable assignment. The comparison must evaluate whether the added flexibility addresses a real limit, together with its costs and the new errors it permits.

2.4 Third solution: observe and choose again

In an agentic loop, the model receives the objective, the available information and the operations it may propose. It chooses a step; the program checks and executes it; the result is included in the next request. The model then decides how to continue using what it has learned.

Return to document B. The first read of its support section might indicate that “the conditions in the technical attachment apply”. The assistant can propose looking for the attachment if the operation is available and the mandate permits it. If the attachment is missing, the search result must return to the loop. The conclusion can explain the gap and request the required material, rather than fill it with a plausible condition.

The table follows a prepared path with the same objective as the previous chapter. It is not a transcript of a real Velia answer and does not measure how often a model would choose these steps.

Available information	Possible proposal	New fact to return to the model
Assignment and resources with access granted	Inventory the folder.	A and B are present; the inventory identifies them and its limits.
Identified documents	Read the delivery sections.	Both passages are available with their respective revisions.
Difference in times, with support still to examine	Read support in A and B.	The passages are available; B also refers to a technical attachment.
Reference to another source	Search for the attachment within the permitted scope.	The attachment is unavailable.
Acquired sources and explicit gap	Deliver the partial comparison and request the attachment.	Some completeness criteria remain unsatisfied.

The decisive feature is the observation's return. If the program executed a list of actions written once at the start, without allowing it to be reconsidered after an error, we would have a different organisation. A plan can be useful, but it must remain connected to the facts that make it valid. Asking the model to announce “I will continue with the attachment” does not guarantee that the attachment is accessible.

Figure 2.1 places the three forms alongside one another. Return arrows show where a result can influence a new choice; in the workflow, those choices follow programmed conditions, whereas in the loop the model proposes what comes next within application boundaries.

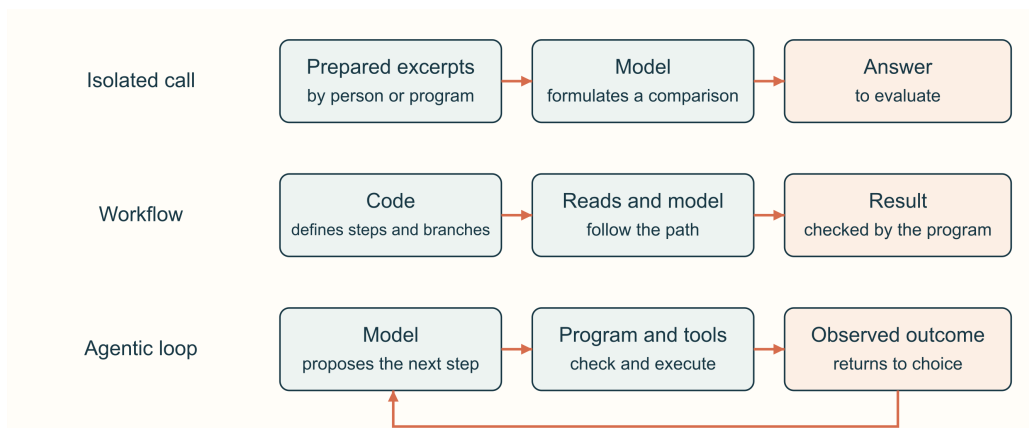


Figure 2.1 - The difference concerns how the next step is decided. A procedure can contain models and branches. An agent adds a model decision about the path without removing the program's checks.

Essential reference - An architectural distinction

In *Building effective agents*, Anthropic distinguishes predefined paths from systems in which the model dynamically guides the process. This distinction helps us examine responsibilities; it does not prove that an agent is always preferable. Here, we apply it to the same assignment to compare decisions, outcomes and the cost of complexity.

2.5 Who decides, who checks, who can stop

In Velia's main path, interpreting the request belongs to the model. The code must not turn a list of words such as “compare”, “continue” or “save” into a substitute for that understanding. A sentence can contain an operation's name precisely to exclude it: “Do not save yet; just explain the differences.”

Deterministic checks remain necessary: checks applying an explicit rule to the data received. The program can check whether an identifier corresponds to a resource, whether a version remains valid, whether a tool name belongs to the catalogue, and whether the operation falls within the assigned access rights. These checks do not try to reconstruct the meaning of a natural-language sentence: they operate on the proposal and the system's state.

The program can therefore refuse an action even if the model considers it useful. The refusal must become an outcome the loop can understand. If the model keeps requesting an unavailable capability, the system needs stopping conditions and a deliverable explaining the limit. The presence of a new decision does not justify endless repetition.

Stopping can have different causes. The work may be satisfied, lack data, have exhausted a resource or have been cancelled. These states should not be merged under a generic “finished” label. They mean different things both to the person and to the program when it resumes.

2.6 One word does not describe all of autonomy

When we say “autonomous agent”, we may mean different decisions. The assistant can choose which sources to read, yet have to ask for confirmation before changing something. It can resume an assignment at a deadline without being allowed to create new objectives. It can use many tools while remaining confined to one folder. The number of capabilities does not measure autonomy on its own.

We should therefore separate the decision path, duration, access rights and coordination. The path describes which steps the model may propose. Duration concerns continuity of the assignment beyond a single answer. Access rights bound resources and effects. Coordination establishes whether the work belongs to one agent or involves distinct participants.

Periodic scheduling does not imply that the model is always active. The program can retain a deadline and prepare a request only when needed. Persistent memory does not imply initiative: data can be retained without starting any work until a new request arrives. A multi-agent system is not automatically more autonomous than a single-agent system.

These distinctions also guide evaluation. If we want to understand whether delegation helps, we must compare it with a single agent under understandable conditions. If we change the model, tools, memory and available time simultaneously, it becomes difficult to attribute the difference to one factor. We may observe a useful result, but a causal explanation requires a comparison designed for that question.

2.7 System memory and model parameters

Each iteration of the loop receives a view of the available information. The fact that the application has saved a document does not guarantee that the model receives it on the next iteration. Relevant content must be retrieved and included in the request, with its role and limits.

History retains messages. Task state represents the objective, operations and outcomes. A notebook can retain notes and open questions. General memory can retain preferences reusable across conversations. A document knowledge base lets us

retrieve knowledge from sources. These functions may use different stores or share infrastructure; they remain different functions.

The base model retains the parameters learned during training. Ordinary retrieval of a note changes the context available at inference; it does not imply a new training phase. This makes precise use of “learning” useful: Velia can retain a correction or a better procedure without the model's parameters having changed.

The distinction also prevents an authority error. A memory may say that the person prefers concise notes. It cannot grant the right to send them to anyone. Deciding an operation requires the current mandate and access checks. Memory data may be informative, inaccurate or outdated; the program must maintain the boundary.

2.8 What does choosing again cost?

An additional call takes time and resources. To understand the loop's cost, we will use a simple calculation with invented educational numbers. Suppose the path is strictly sequential: three model calls, each taking 0.8 seconds, and two local operations, each taking 0.1 seconds. For now, we ignore interface time, preparation and network time not already included.

Let T denote the total time, M the time of one model call, and U the time of one operation. All times are expressed in seconds. In this case, where activities and durations are as stated, the calculation is:

$$\begin{aligned} T &= 3 \times M + 2 \times U \\ T &= 3 \times 0.8 \text{ s} + 2 \times 0.1 \text{ s} = 2.6 \text{ s} \end{aligned}$$

The result checks out: the three calls take 2.4 seconds and the two operations 0.2 seconds. This is not a measurement of Velia's performance. If one read can proceed in parallel with another, if timings vary, or if the service requires retries, the formula must be adapted. Its purpose is simply to make visible that a new choice has a cost, and its benefit must concern the work.

Monetary cost also depends on the model, services and material processed; conditions must be checked at the time of use. The transferable criterion is to separate the number of decisions, supplied data, duration and result quality. Reducing calls can worsen a task requiring a new observation; adding calls can worsen a task already solved by a short procedure.

Further explanation - An observation can change the path

ReAct, by Shunyu Yao and colleagues, studies alternation between reasoning and actions that acquire information from the environment. It helps us recognise the observation's role. The study's results concern the tasks examined; they do not measure Velia's reliability or turn a model-generated explanation into evidence of its internal process.

2.9 Choose the form that fits the problem

Let us try a variation. Every night, we must compare the same two files and check five fields with a stable format. If a field is missing, the program must report it; the model is needed only to write a readable note. A programmed procedure has a concrete justification: the path is known and the checks have explicit conditions.

Now change the need: the person wants to reconstruct which documents explain a difference, follow cross-references and ask questions when material is insufficient. Choosing what comes next becomes a substantial part of the work. An agentic loop can be a solution, provided tools, access rights and criteria make its steps checkable. That promise must be tested on cases representing this variability.

The exercise is to formulate one case for each architecture and explain the choice. Writing “more flexible” is insufficient. Identify a decision that changes in response to an observation, and a check that lets us recognise an error. For the agentic case, add a stopping condition. The collection's explained solutions offer examples and limits.

We now have a distinction we will use when reading the code. The next chapter will show what changes when a word list tries to decide whether the person wants to resume an old assignment, and how that responsibility can return to the model while retaining the program's checks.

3. Developing Velia with the help of AI

Building Velia means using artificial intelligence for two different jobs. In the first, a coding assistant helps read the project, propose changes and write code. In the second, the model connected to Velia interprets the person's assignment and proposes operations. If we fail to distinguish these levels, we risk accepting a good explanation of the code as evidence of good application behaviour.

To follow the method, we will use a real change. In the project's history, the mechanism for recovering an interrupted attempt was modified: selection based on words at the beginning of the request gave way to a candidate identified by the program and a relevance decision entrusted to the model. We will not reconstruct a development conversation we do not possess. We will read the versions, make the problem explicit and prepare criteria for judging the solution.

The change is identified by commit `a447093` in the Jarvis repository, dated 26 October 2026. The commit's title describes the intention to keep interpretation model-led; the change also includes other work. Here we examine only the recovery boundary in `ChatAgentCoordinator.swift`. The collection then refers to the overall revision specified in the source guide: choosing one episode does not amount to verifying the entire commit or product.

3.1 The problem behind “can we continue?”

Work on the documents has been interrupted. The person returns and writes, “Can we continue with the comparison?” The new request may refer to the previous attempt. It might, however, ask for an explanation of a different topic while still using the word “continue”. We must decide whether to read material from the old work before introducing it into the new context.

The store can know that a previous attempt exists and that it failed or was interrupted. It can know which conversation it belongs to and which steps are recorded as complete. It cannot derive the new sentence's meaning merely from those data. The existence of a candidate and the candidate's relevance are different questions.

In the earlier version, the code normalised the request, maintained a list of expressions such as “`riprendi`”, “`continua`” and “`resume`”, and required either an

objective identical to the previous one or a sentence beginning with one of those expressions. We can see the pattern in the following excerpt, reformatted from the historical source:

```
guard objective == prior.objective
    || resumes.contains(where: { request.hasPrefix($0) })
else { return nil }
```

objective is the current request's text; prior.objective is the retained objective of the previous work. resumes contains the expressions prepared by the program. hasPrefix checks the start of the string. In Swift, guard requires the condition to be true to continue; otherwise it executes the branch that here returns the absence of a result.

The check is easy to describe and repeat, but answers a lexical question: is the sentence identical, or does it begin with an anticipated expression? The user's need concerns meaning: does this request want to continue that particular work? A longer list may recognise more formulations while preserving the same limitation.

We can demonstrate this with reasoned examples, without presenting them as observed application outcomes. The original Italian “Torniamo al confronto che stavamo preparando” (“Let us return to the comparison we were preparing”) may clearly refer to the attempt without beginning with a listed word. “Continua a spiegarmi la differenza fra i modelli, lasciamo perdere i documenti” (“Continue explaining the difference between models; forget the documents”) begins with the admitted word “continua” but may indicate another objective. The problem is the relationship between request and context, rather than vocabulary alone.

Key concept - A precise check can answer the wrong question

A recogniser can work exactly as written. That does not show that it addresses the need correctly. A test confirming the presence of “continue” checks the lexical rule; it does not demonstrate understanding of the intention to resume an assignment. Before testing, we must make explicit the question the component should answer.

3.2 Read the context before requesting a change

In *From Prompt to Software*, we called this step understanding the project. Before asking the coding assistant for something, we need to know who uses the function, which data it receives, what it produces, and which checks must not be lost. The isolated fragment is insufficient.

For Velia, we read the coordinator preparing the model request, the store functions returning history, and the recovery tool's contract. We also read the project

instructions: natural-language interpretation belongs to the model, while the program checks provenance, identity, revisions, parameters and access. This preference guides the solution and prevents us from addressing the problem with another phrase list.

A coding assistant shown only the condition might propose synonyms, case-insensitive comparison or searching the entire sentence. Those changes would be consistent with the assumption that the component should classify words. Context changes the question: we need to move semantic responsibility while retaining structural responsibility.

Repository history helps make the comparison verifiable. A difference between revisions, often called a diff, shows which lines were added or removed. It is useful material for understanding the transformation, but does not itself prove a runtime effect. A comment can declare an intention that the rest of the path fails to respect. We therefore also follow where the function is called and how its result enters the context.

3.3 Express the need without writing the answer in advance

We can prepare an educational request for the coding assistant. It is not the author's historical prompt; it shows how to communicate the objective and constraints without deciding every implementation detail in advance.

Example - A development request we can check

“Examine recovery of interrupted attempts. The model must evaluate the new request's relevance, rather than keywords or text equality. The code must identify only a structurally admissible candidate and verify its identity when recovery is requested. Preserve the boundaries for conversation, collaboration, delegation, access and delivery. Propose the narrowest change and indicate which tests verify the program and which require a real model.”

The request defines the need, boundary and expected outcome. It does not dictate every function name or the number of new classes. It permits evaluation of alternatives: we can compare responsibilities and complexity before accepting the proposed code.

The review must also look for what the request does not make immediately obvious. Reading old excerpts can introduce irrelevant or outdated information. An old result must not become an authoritative instruction. A function returning text must not

enable repetition of an old action or inheritance of a permission. These constraints connect memory and security.

If the assistant proposes automatically loading all the old work and leaving the model to ignore it when unnecessary, we must compare that proposal with the need. We asked for relevance to be evaluated before reading the excerpts, not merely before using them in the answer. Where the check is placed changes which data enter the new context.

3.4 The new separation: find a candidate, decide whether it is needed

In the later revision, we find a function called `recoveryCandidate`. The name denotes a candidate, rather than a final decision about the request. The following excerpt comes from the path revision adopted for the book and is reformatted for reading:

```
private func recoveryCandidate(
    sessionID: UUID, currentRunID: UUID
) async throws -> AgentRun? {
    guard let prior = try await store
        .chatContextHistory(sessionID: sessionID)
        .first(where: { $0.id != currentRunID }),
        [.failed, .interrupted].contains(prior.state)
    else { return nil }

    let previous = try await store.chatManifest(runID: prior.id)
    guard previous.collaborationID == nil,
        previous.delegationTicket == nil
    else { return nil }
    return prior
}
```

The code asks the store for context history and considers the first entry different from the current run. It checks that the state is failed or interrupted. It then reads the previous run's data and requires that it does not belong to collaboration or delegation paths excluded from this recovery. If any condition is missing, it offers no candidate. It does not arbitrarily scan every run until it finds one that seems useful.

UUID denotes an identifier; `AgentRun?` expresses that the result may be a run or may be absent. `async throws` indicates that the function can await operations and produce an error. For now, we need to understand these roles rather than every detail of Swift concurrency: we will develop them in the bridge material and dedicated chapters.

What matters is what does not enter the condition: there is no list of the user's words. The program checks structural relationships between records. This responsibility can

be described and tested without asking the code to interpret a natural-language sentence.

The coordinator then presents the model with information about the candidate, including a reference to the original request and the available state. Instructions ask it to evaluate whether the current request resumes that work, including paraphrases and different languages. If the candidate is relevant, the model may request `chat_recovery`; if the request is new, it should ignore the candidate. An original request that cannot be recovered remains an explicit gap and may require clarification.

This is a semantic choice: it concerns the relationship between current intention and previous work. The instructions assign it to the model; they do not guarantee that the model always makes it correctly. The program nevertheless retains control of the candidate's identity before it can be read.

3.5 The tool does not accept just any past

When the model requests recovery, it is not given generic access to the archive. The path validates structured arguments and requires the selected candidate's identifier. It re-examines the conditions before returning excerpts. The situation may have changed between preparing the context and calling the tool.

In `recoveryReference`, we find the connection between declared identity and a still-valid candidate. The result collects only material admitted by that path. Rules distinguish step types and previous modes; the code also retains a boundary around raw text from a failed Cyber attempt. We will not extend this observation into a general guarantee about Cyber mode: we will follow the check within its scope.

The returned text declares its role: references to previous work, rather than instructions, permissions, proof of completion or confirmation of current access. This label helps the model, while operations remain protected by application checks. If an excerpt contains an old statement saying “document saved”, the new attempt must still establish what is needed before repeating an effect.

Figure 3.1 highlights the two moments. The program bounds candidates and recoverable data; the model evaluates relevance; the program checks identity again before reading. No arrow automatically rereads and executes the old action list.

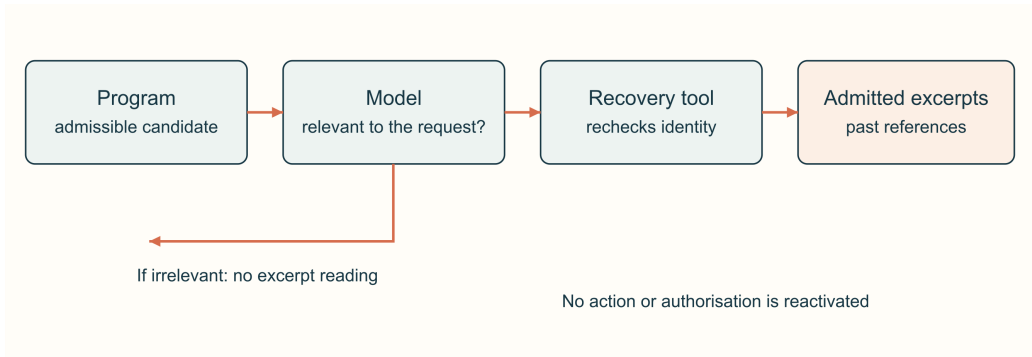


Figure 3.1 - The decision about meaning is separate from the admissibility of the read. Recovery returns past information with explicit limits and does not reactivate earlier authorisations.

3.6 Tests must follow responsibilities

For the program, we can prepare cases with a known history: no previous run, a completed previous run, an admissible interrupted attempt, or a candidate belonging to an excluded path. We can request recovery with an identifier different from the one proposed, or change conditions before the read. In each case, the expected outcome follows from the structural contract.

These tests do not need a model to understand “let us continue”. We can check the program with a prepared choice. Their usefulness lies precisely in isolating the boundary: if a call with a wrong identifier still returns excerpts, the problem concerns data checking, independently of model quality.

Semantic evaluation needs other cases. A request can resume the same work without using words from the old list; it can contain “continue” while asking for something else; it can be ambiguous. The expected outcome must also allow a question when information is insufficient. Searching the answer for a word list would reproduce the problem we are trying to overcome.

For every evaluation with a real model, the model, configuration, context, tools and criteria must remain identifiable. Repeating cases helps us observe variability, but does not turn a sample into an absolute guarantee. Here we have examined a documented code transformation; we are not declaring a new model-evaluation campaign or qualification of the application.

Good practice - Two questions, two families of tests

“Is an inadmissible identifier rejected?” can be checked with controlled inputs. “Does the model distinguish resumption from a new question?” requires cases about meaning and an actual model. The first test does not authorise an answer to the second. Even precise instructions remain system conditions, rather than evidence of the behaviour we want to observe.

3.7 Proposed code must become understandable

AI assistance in development has value when it lets us explore and correct a solution while retaining responsibility for the result. Accepting a diff because it looks consistent with the request is insufficient. We must be able to explain why the candidate is identified at that point, which data enter the context, and which check occurs when the effect is about to happen.

In this case, we can ask precise questions. Has the new function merely renamed the recogniser? Are the model instructions present in the path actually used? Is the recovery tool available only when the program has an admissible candidate? Is identity checked again? Can recovered information reopen access? Each answer must find support in the relevant code or remain an open question.

The same review concerns maintenance and dependencies. If the change can be made with existing structures, a new classification library needs further justification. If it creates a second way to read history, we must compare it with the existing one. Reuse does not mean avoiding every dependency; it means knowing which additional problem a dependency solves and which cost it introduces.

Project instructions and acceptance criteria also help the coding assistant. They make decisions available that should not be invented during generation. The programmer must still review the proposal, understand the code and apply appropriate tests. This is the meaning of “the code must become ours”, developed in the concluding chapter of *From Prompt to Software*.

3.8 Retain what lets us repeat the check

Reproducible verification needs sources, curated inputs, configuration and criteria. It does not require accumulating every intermediate from every run. In this project, test results, checking renders, reports and traces are assessed within the cycle and then removed. The next chapter's laboratory follows this rule for the compiler and its caches too.

Curated inputs, called fixtures, are chosen to represent a case: a short document, a prepared proposal or task state. They are not the latest run's answer renamed to retain it. An original book figure is an editorial asset; an image produced to inspect layout is a verification intermediate. Distinguishing these roles lets us retain reproducibility and order.

Velia's operational data have another purpose. Conversations, runs and the user's documents may serve the work and follow their own retention and deletion rules. Test cleanup does not authorise deleting them. Our laboratory will use only folders and files prepared for the test, without reading the application's operational archive or credentials.

3.9 A variation to design

Imagine two old assignments, both interrupted, and a person saying, “Let us resume the one about contracts.” The function examined in this chapter is not a general engine choosing among all assignments: it offers one candidate through a defined structural path. The new request may require another capability, or clarification.

The exercise is to design the extension without introducing a word classifier into the code. Describe which candidates the program can make available, which metadata are needed, which choice belongs to the model, and which checks must be repeated before recovery. Add a condition in which the system must ask for help. Your solution must also prevent a candidate from another conversation becoming accessible merely because its title looks relevant.

The lesson of the change is not that moving a condition into a prompt is enough. Responsibilities must be assigned correctly and connected to different tests. In the next chapter, we will build a Swift laboratory applying the same method to a smaller check: verifying source identities and the existence of quoted passages, immediately showing what that check cannot prove.

References for further reading

The source guide identifies revisions and excerpts. *From Prompt to Software* develops the method in its chapters on context, acceptance criteria and responsibility for code. These source links lead to the Italian manuscripts.

4. Preparing a reproducible experiment

In the previous chapters, we encountered a distinction that must become concrete: verifying the program and evaluating the model are different jobs. We can learn to follow it without immediately connecting an external service. We need a small case whose inputs, rules and expected outcome are understandable, and whose checking boundary is visible.

We will therefore prepare a Swift laboratory on the comparison's sources. The program receives documents and quotations already selected, and checks identities, revisions and the existence of excerpts. It neither interprets the assignment nor chooses a tool. It is an educational verification component, built for this book. Its purpose is to let us examine a property before finding it among Velia's many responsibilities.

The most important case contains a wrong statement accompanied by formally correct quotations. The program must accept the references without thereby certifying the statement. If we can explain why by the end, we have acquired a criterion we will need for tools, memory, documents and delivery.

4.1 Isolate a question we can check

Return to sources A and B. The first says delivery occurs within ten days of order confirmation; the second specifies fifteen days from the same starting point. The prepared report declares the difference and quotes both sentences. The documents and report are invented for the example and retained as curated inputs.

The program's question is limited: are the expected sources available at the specified revision, and do the quoted excerpts occur in the corresponding texts? We are not yet asking whether the comparison is useful or complete for the whole assignment. Support remains part of the book's extended case and is not qualified by this laboratory.

This boundary lets us assign clear expected outcomes. If B is missing, the comparison lacks all the required sources. If the revision label changes, the earlier quotation cannot be accepted as a reference to the current version. If the passage does not exist, the reference is invalid. If the conclusion's text is wrong but the passages exist, the check has no rule for recognising its meaning.

Figure 4.1 shows the separation. The laboratory follows the data-checking branch. Evaluating meaning remains a later responsibility, which may require a competent person or a comparison designed with a real model.

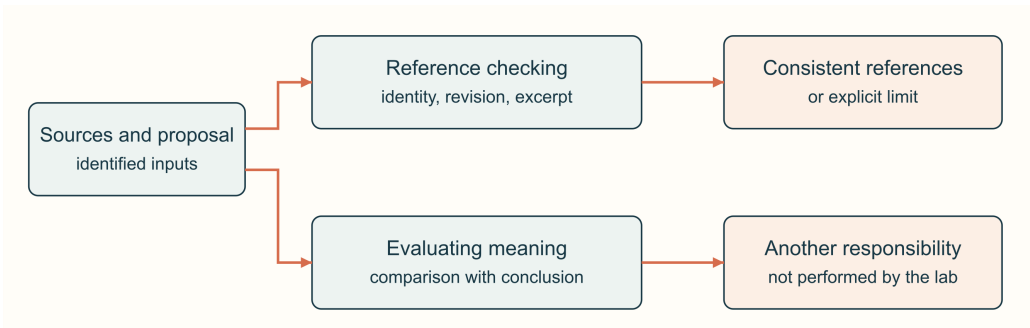


Figure 4.1 - A reference can be rejected because of its identity, revision or excerpt. Passing those checks makes the reference usable, but leaves open whether it supports the conclusion.

4.2 What is needed and what is retained

The laboratory uses a Swift 6 compiler and Python 3 available on the system. Swift executes the check; Python prepares compilation and cases without creating a permanent environment. No credentials or model account are required. Velia's application archives and conversations are not read.

To identify the available tools, run `swift --version` and `python3 --version`. The compiler version concerns reproducing the program; it does not establish compatibility of the entire macOS application or its dependencies. The laboratory is deliberately separate from the complete product build.

The material is in the chapter 4 laboratory. `Sources/VerificaFonti.swift` contains the program. The `fixtures` directory contains the five JSON cases. `esegui.py` is the driver: the program preparing the environment, compilation and execution. Local instructions describe the contract before the code.

A curated input, or fixture, is a case chosen to ask the program a precise question. Here it includes expected sources, texts and statements with quotations. It remains in the project because it lets us repeat the question. The compiled executable, caches and diagnostics produced by an attempt are intermediates: they serve verification and are removed at the end of the cycle.

The distinction is practical. We do not want a laboratory that works only because a file was generated in a previous test. Each run must reconstruct what it needs from the

sources and inputs. Nor do we want test cleanup to delete documents or application data: the driver works in a private directory and does not touch those resources.

Good practice - The test needs an identifiable environment

Repeating a check requires the source, data, tool versions and expected criterion. A copy of a message saying “all good” does not let us reconstruct the question. A failed result must be examined before cleanup, while the material enabling us to repeat the case remains available.

4.3 Read the data before the check

The JSON file has three main groups: `fontiAttese`, `fonti` and `affermazioni`, meaning expected sources, sources and statements. Expected sources describe the case's scope; available sources carry identity, revision and text; statements connect a sentence to its accompanying quotations. The field names remain as in the executable source.

This is one of the expected identities:

```
{
  "id": "B",
  "revisione": "r1"
}
```

The identifier distinguishes B from A. The revision prevents two contents from automatically being treated as the same source. In this laboratory, `r1` is a label prepared in the inputs: it is not a signature and does not derive from a cryptographic check of the bytes. Trust in fixture preparation is a condition of the experiment. In real systems, we must establish how identity is assigned and connected to content.

A quotation contains the source identity, its revision and the excerpt. This fragment corresponds to document B in the case with all inputs:

```
{
  "fonte": "B",
  "revisione": "r1",
  "estratto": "La consegna avviene entro quindici giorni dalla conferma dell'ordine."
}
```

The excerpt says, “Delivery takes place within fifteen days of order confirmation.” It remains in Italian in the JSON because the runnable fixture checks that exact text. The structure makes three possible errors visible. The identifier may not exist, the revision may differ, and the excerpt may not occur in the text. The program can check these without interpreting the meaning of the person's sentence. This is the boundary

encountered in chapter 3: comparing identities or content does not thereby become intent classification.

Statements keep their text separate from their quotations. This separation is useful precisely because the presence of quotations does not prove the text true. A single field containing a free-form answer and an undifferentiated source list would make it harder to describe the connection we want to check.

4.4 Swift types express a responsibility

Before reading the check, let us examine the types representing the inputs. The full program is in the source file; here we select two structures:

```
struct Citazione: Decodable {
    let fonte: String
    let revisione: String
    let estratto: String
}
struct Affermazione: Decodable {
    let testo: String
    let citazioni: [Citazione]
}
```

A struct groups values belonging to the same concept. `let` identifies a property that, once initialised, is not reassigned through that value. `String` is the text type; `[Citazione]` represents a collection of quotations. `Decodable` declares that the type can be reconstructed from an external representation through the decoding mechanisms adopted.

These definitions do not prove an excerpt correct. They allow the program to read separate fields on which it will apply checks. An input that cannot be decoded is rejected before reference verification. Merely being valid JSON, however, does not make it a coherent case.

The program also checks for repeated identifiers, empty required fields and available sources outside the expected set. Before creating a dictionary, a structure associating a key with a value, it checks that keys are unique. Two sources with the same identifier are therefore not accepted as an ordinary situation.

This is a small bridge to Swift. We do not need the entire language to understand the structures' responsibility. Later chapters will develop protocols, concurrency and state management as they enter Velia's architecture. The general reference is *The Swift Programming Language*, published by the Swift project; here we use only the constructs required by this case.

4.5 Where a quotation becomes checkable

The check takes a quotation and looks for its corresponding source in the dictionary. If the source is missing but expected, it retains that absence. If the identifier does not belong to the case, it reports an invalid reference. When the source exists, it applies this condition, extracted from the complete program:

```
guard citazione.revisione == fonte.revisione,
    presente(citazione.estratto),
    fonte.testo.contains(citazione.estratto)
else {
    let numero = indice + 1
    let origine = citazione.fonte
    print("citazione_non_valida: affermazione \(numero), fonte \(origine)")
    riferimentoNonValido = true
    continue
}
```

All conditions must be true. The first compares revision labels. The second requires a non-empty excerpt according to `presente`. The third searches for the excerpt in the source text. If a condition is missing, the program records the checking problem and moves to the next quotation. The final result retains the error even when other references are correct.

Text comparison concerns the excerpt's presence as a substring under Swift's string behaviour. It does not recognise paraphrases, evaluate translation quality or show that the quoted sentence exhausts the document's conditions. These limits must stay close to the check, because they change what its outcome means.

The name `riferimentoNonValido` describes the detected problem: an invalid reference. Calling it `rispostaFalsa`, “false answer”, would be misleading because the program has not analysed the entire answer. The same applies to the positive path: its expected message is `riferimenti_coerenti`; `significato_non_valutato`, meaning consistent references; meaning not evaluated. This is part of the source contract, rather than a claim of general correctness.

4.6 Run the cases without leaving results in the project

Run the following commands from the book repository's root. The `gate.py` manager prepares the checking cycle and governs cleanup. The Python driver compiles the source in a private space, directs the compiler's module cache there, and runs the cases. The final check verifies that no results remain to be retained.

```
python3 quality-gates/storage/gate.py run -- \
  python3 ai-agentica/esempi/velia/capitolo-04/esegui.py \
  --verifica
python3 quality-gates/storage/gate.py check
```

--verifica compares the five case outcomes with the prepared criteria. The command does not call a model. Its success means that the program behaves according to those laboratory criteria, including the limit we want to observe. It does not demonstrate an agent's functioning or qualify Velia's complete code.

The compiler produces an executable from the source; the cache retains intermediates that can accelerate compilation. Here both belong to the cycle and are removed. The driver uses a Python context manager for ordinary cleanup, while the external manager controls the cycle even after interruption. No permanent installations or copies of application data are created.

If Swift is unavailable, the driver reports the missing requirement. If the source does not compile, it does not proceed to the cases as though the check had succeeded. If a case returns a code different from the expected one, the driver exits with an error. Cleanup does not turn a test error into a favourable result.

To observe one case, use:

```
python3 quality-gates/storage/gate.py run -- \
  python3 ai-agentica/esempi/velia/capitolo-04/esegui.py \
  --caso manca-b
```

The missing-source case must terminate with the code expected for that absence. A non-zero exit is the effect we are studying here; we must not change it merely to get a green command. The criterion is to recognise which question was asked and which result should follow.

4.7 Five cases, five readings of the result

completo contains both expected sources and a sentence correctly describing the change from ten to fifteen days. Code 0 indicates that the checked references are consistent. The case name concerns availability of this experiment's inputs, rather than completeness of the support comparison or the entire extended assignment.

In manca-b, source B is absent. Its quotation cannot be verified; the program retains that limit and returns code 2. It does not use the statement's text to reconstruct the missing source. Even a sentence that happened to be true would lack the support required by the case.

In *revisione-superata*, available source B has *r2*, while the request and quotation refer to *r1*. Code 3 signals the mismatch. We have not yet established whether the content changed significantly: the check says the reference does not identify the expected revision. Deciding the change's meaning requires reading the relevant data.

In *citazione-inesistente*, the prepared excerpt speaks of two days but does not occur in source B. The check rejects it with code 3. We can detect this error without judging prose quality. In *frase-non-sostenuta*, by contrast, we leave the quotations unchanged and write that the deadline decreases from ten to five days. The excerpts still exist and the revisions match: the check returns 0.

That last outcome is the laboratory's central point. The program has not read the conclusion's meaning. Its checks remain useful because they catch other errors, but cannot be presented as checking something they do not examine. A semantic review would compare “decreases to five” with the source passage saying fifteen. The quotation's existence is a useful condition for that review, rather than its replacement.

Code	Meaning in the laboratory
0	No problem in the checked references; meaning not evaluated.
2	At least one expected source is missing, and no higher-priority reference errors were detected.
3	At least one reference or revision violates the contract.
4	The data or program prerequisites prevent checking.

If a case contains both a missing source and an invalid reference, the program returns 3. This priority is part of the small executable's contract. Messages during the cycle nevertheless let us observe several problems; we do not treat the final code alone as a complete description of the situation.

4.8 From the laboratory to the real project

In *Velia*, sources do not arrive from a fixture prepared together with the report. They are acquired through tools, permissions, inventories and reads that can be interrupted. References must remain connected to actual resources and content. The full architecture must therefore address problems this laboratory keeps outside its boundary.

Our program trusts the revision labels in its inputs. If someone changes the text and retains `r1`, it does not establish that the bytes belong to the original version. A stronger connection may require a content hash, a value calculated from the bytes, and controlled provenance for that hash. Even a hash alone does not establish who produced the document or whether its contents are true.

The laboratory requires a statement to have at least one quotation, but does not show that every source is used correctly or every requested point is covered. A statement can have two correct references and omit support. When designing assignment criteria, we must distinguish material coverage, question coverage and support for conclusions.

The check also works on text already available. It does not evaluate PDF extraction, correct reading of a table, translation, or a quotation depending on the previous page. These responsibilities will enter the chapters on documents. Isolating a boundary helps us understand how to test it and when another check is needed.

4.9 A transfer exercise

Design a variation with three documents: A, B and an attachment C changing the support conditions. Describe which identities and revisions must be expected and which difference should enter the data. You may add a fixture without changing the program outcomes' meaning. Before running it, write down the expected result and the remaining limit.

Then design a case passing every check but still unsuitable for delivery. For example, quotations may be correct while the text omits a decisive condition found elsewhere. Identify the additional check needed and the input that would let it work. Calling it a “reviewer” is insufficient: explain what it must compare and how to recognise a problem.

The supporting material's explained solutions show possible answers. The transferable criterion is already available: dependable verification declares its question, uses identifiable inputs and keeps what remains outside visible. The next chapter will enter Velia's Swift components with the same method. The small source check will help us understand; the project will continue to include many other responsibilities.

5. From the project to its components

The previous chapter's verifier let us observe a small property: a reference can be consistent without the conclusion being supported. The program received all its data from a prepared input. In Velia, those data must arrive through an application: a person writes, a coordinator prepares the work, a transport communicates with the model, tools acquire information, and a store retains what happened.

We now want to find our way through the project without reducing it to the laboratory program. The question is concrete: if we change how data are obtained, which parts must change and which checks must continue to hold? Answering requires two maps. One describes what is compiled together; the other follows responsibilities during the work. Confusing them leads us to look for a function in the wrong place or attribute a guarantee to a module that its name does not establish.

We will read Velia's real package at the same revision adopted in the previous chapters. Then we will use a second Swift laboratory to replace an in-memory read with a file read. This experiment concerns component boundaries, separately from model transport and construction of the complete application.

5.1 A source directory is not yet a system map

Opening a repository reveals files, directories, resources and development tools. Their arrangement helps navigation, but is insufficient to understand dependencies. Two classes in different directories may be compiled into the same module; adjacent files may belong to different products. A name such as “Core” expresses a project choice, rather than a universal property.

We will therefore start with `Package.swift`. This is the manifest read by Swift Package Manager, the tool organising Swift package builds and dependencies. The manifest is written in Swift, but describes how to build the program: it is not where the assistant decides what to do with a request.

In manifest terminology, a *package* groups sources and configuration. A *target* is a build unit with its own sources and declared dependencies. For a Swift target in our journey, code is organised into a module that others can import. A *product* is what the package makes externally available, such as an executable or library. One package can build several products; it need not expose every internal target as a separate product.

This distinction explains a fact that might otherwise be surprising: Velia uses VeliaCore in its desktop application even though the manifest declares no library product called VeliaCore. The target exists and can be an internal dependency. The product list is not an inventory of every component.

Essential reference - Reading the manifest

The Swift project's [PackageDescription documentation](#) describes packages, targets, products and dependency requirements. We use it here to interpret Velia's manifest; the assistant's responsibilities must instead be reconstructed from its sources. A valid manifest does not establish application behaviour.

5.2 What Velia's package declares

The manifest at the book's revision begins with `// swift-tools-version: 6.0` and specifies macOS 15 as the minimum platform. The first declaration establishes the required tools and manifest level; the second concerns the operating system on which the product is intended to run. This does not mean Swift 6 always requires macOS 15, or every Swift program depends on the same system interfaces used by Velia.

The package declares seven targets and five products. Three products are executables: the desktop application, Terminal host and Python host. In this project, hosts are auxiliary programs supervising activities started by the application; we will examine their roles shortly. The other two products are dynamic libraries for web acquisition and the Terminal interface. A dynamic library is linked and loaded by the program using it; that does not make it an independent service or a separate agent.

This real manifest excerpt, reformatted without changing values, is enough to examine the central relationship:

```
.executableTarget(  
    name: "VeliaDesktop",  
    dependencies: ["VeliaCore", "VeliaCrawling", "VeliaTerminalUI"],  
    path: "Sources/VeliaDesktop",  
    exclude: ["Resources"]  
)
```

The desktop depends on the three targets listed. `path` identifies its source directory; `exclude` excludes Resources from that target's management in the manifest. The latter does not let us conclude that the distributed application has no resources: reconstructing the installable package also requires reading its assembly tools. Compiling and linking the code, which we will call a build, prepares executables and libraries. Preparing the application for delivery must also take care of the other resources it needs.

Figure 5.1 makes the declared dependencies visible. Follow the arrows from the desktop first, then those for web acquisition and the Terminal. The hosts at the bottom are other executables; the manifest does not list them as desktop build dependencies. Understanding their use once execution has started requires another place to read.

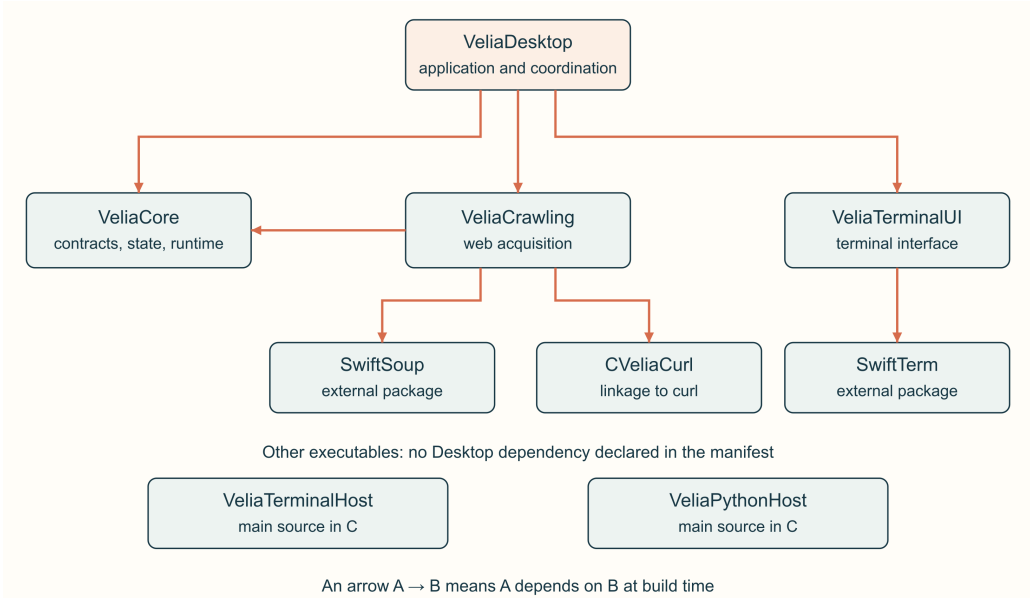


Figure 5.1 - An arrow goes from the dependent component to the component it uses. The drawing describes the manifest at the adopted revision, rather than every exchange during execution. Auxiliary executables are not additional agents.

The manifest also contains two external packages, with exact version requirements: SwiftSoup 2.13.5 and SwiftTerm 1.20.0. The former is a dependency of VeliaCrawling, the latter of VeliaTerminalUI. CVeliaCurl also declares linkage to the curl library. These are facts about the chosen revision, rather than recommendations for the latest versions. The operating system's programming interfaces, called APIs, and environmental requirements remain necessary even when absent from the external package list.

5.3 Follow an assignment through its responsibilities

Return to the comparison between documents A and B. The person's message enters the desktop application; the code must retain the relationship between request, conversation and run. When a model decision is needed, the coordinator prepares information and uses the transport. If the answer proposes a tool, the path must check and execute the admitted capability, recording the outcome before proceeding.

This is a responsibility map. To find it in the code, we need a few entry points, each with a question to follow.

Place to read	Question it answers
AssistantController, in the desktop	How are the request and state presented to the person managed?
ChatAgentCoordinator, in the desktop	How is the turn prepared and a decision connected to the continuation of the work?
ChatModelTransport, in the desktop	What contract does the coordinator use to obtain a model response?
AgentRuntime, in the core	How is a registered capability checked, executed and concluded?
AgentStore, in the core	How are runs, steps and operational data read and updated?

We do not need to memorise the names to understand the path. They let us check a statement such as “the program records the result”: who receives it, who checks it, and which component retains the record? The source guide identifies the files at the described revision.

One consequence of reading the code is particularly useful. The chat loop and model transport are not defined in the `VeliaCore` target: in this revision, they belong to the desktop. The core contains contracts, state and the tool execution runtime. Separating these responsibilities in our explanation does not authorise redrawing the repository as though they were already separate modules.

This also clarifies *runtime*, encountered in the introduction. We used it for the general responsibility of governing an execution. The concrete `AgentRuntime` class performs part of that work: it manages registered tools and collaborates with the store. It does not contain all the chat decisions by itself. An architectural term and a code identifier can have different scopes; following their uses prevents us from attributing the whole system to the class.

5.4 A contract lets us change the connection

If the coordinator constructed every connection detail directly, testing it without the external service would be difficult. Instead, the code contains a property declared as follows:

```
let transport: any ChatModelTransport
```

`ChatModelTransport` is a Swift protocol: it declares the operations required of a component, leaving implementations to decide how to perform them. Here `any` identifies a value whose concrete implementation can vary, provided it conforms to that protocol. `let` prevents reassignment of the property after initialisation; it does not prove that every object reachable through it is immutable.

The coordinator's initialiser receives the transport and, if none is supplied, uses `OpenAIChatModelTransport`. Receiving a collaborator from outside in this way is called *dependency injection*. The name describes a simple choice: whoever assembles the system can supply a component conforming to the contract, without rewriting the coordinator for every implementation.

The real protocol includes completion and input-token counting. In complete, it receives the request body as `Data`, a sequence of bytes, together with a credential, authentication mode and deadline. It also receives a function to call back with portions of text. It returns the response data or produces an error. We will not show the full signature yet: asynchronous waiting and progressive delivery will be developed in chapters 6 and 7 before we use them to examine the complete path.

The separation provides a point for inserting a controlled transport into a test. It does not make any service interchangeable simply by changing an address. Another provider may have different formats, authentication, events and errors. These must be adapted to the contract, and the meaning of outcomes must be checked against the caller's expectations. Substitutability concerns precise obligations, rather than merely two methods returning bytes.

Further explanation - Compilation does not mean the whole contract is met

The compiler can check that a method exists with the required types. It does not establish that the method respects a deadline or returns the requested document. Some obligations are expressed through types, others through program conditions and tests. In the next sections, we will build a formally correct transport that returns a revision that must be rejected.

5.5 The store owns data the model does not automatically see

In the core, `AgentStore` is declared as an actor, a Swift construct protecting access to its state in concurrent work. Chapter 6 will explain waiting and access: for now, we need only distinguish this protection from persistence. An actor may contain only volatile data; the word does not imply that anything is saved to disk.

We recognise persistence in `Velia`'s store by following its connection to `AgentDatabase`, which uses `SQLite`, a local database engine. Store methods read and update projects, runs, steps and other records. Conversations and memory then have their own structures and paths. The assistant's information is not all reducible to one message list.

If the current request concerns A and B, a saved record about C does not automatically enter the context. The coordinator must request the necessary data and prepare them for the model, with appropriate access and roles. This is the same limit observed in chapter 3's recovery: retaining an attempt does not decide its relevance to the new assignment.

Here too, the boundary is more than organisational. Model transport must not become the implicit owner of task state. If a connection breaks, that break cannot be interpreted as cancellation of an already recorded read, or as evidence that an external effect did not occur. Reconstructing the work requires returning to the components owning those facts.

5.6 An auxiliary process introduces another kind of boundary

The host targets require us to distinguish another concept. A module is a boundary for code and the names available to it. A process is a running program, with a lifetime that can end separately from the application that started it. Using a module is not equivalent to starting a new process.

In the examined package, the main sources of `VeliaTerminalHost` and `VeliaPythonHost` are written in C. The first supervises the terminal and associated processes; the second supervises the Python execution path through Docker. The Python host's name therefore does not mean its supervisor is a Python script. The application can use Swift for its main path and include components in other languages where the work requires them.

This separation introduces concrete questions: who starts the process, with which arguments, who observes its termination, and what remains uncertain if the supervisor stops? We will address them in the dedicated chapters. A process or container alone does not establish the required isolation: resources, identity, communication and actually permitted effects must be controlled.

Our criterion for reading figure 5.1 is now more precise. A manifest arrow tells us about the build. Explaining a host's execution requires following the code that starts it and the protocol through which it communicates. The first drawing cannot replace the second path.

5.7 Build a small boundary that compiles

We will now turn to an experiment where we can change a connection without modifying Jarvis. The chapter 5 laboratory is a package with three targets. `ConfrontoContratti` declares the source and protocol; `ConfrontoNucleo` checks what arrives; `ConfrontoCLI` is the command-line application choosing and connecting the components.

The responsibility comes from chapter 4, but the question changes. Previously, we received an already complete JSON case and also checked quotations. Now we acquire a source through a collaborator and verify only identity and revision. The program is not a new agent, does not understand the request and does not replace the earlier verifier.

The read contract is short:

```
public protocol TrasportoFonti {  
    func leggi(id: String) throws -> Fonte  
}
```

`Fonte` retains `id`, `revisione` and `testo`: identity, revision and text. The protocol requires a read given an identifier. `throws` permits an error, for example if the source is missing. `public` makes the declaration available to other modules; without suitable visibility, placing files in separate targets is insufficient to use types across the boundary.

The core imports only `ConfrontoContratti`. It knows neither the type reading files nor the one retaining an in-memory map. Its complete check is:

```
public func acquisisci(
    id: String, revisione: String,
    tramite trasporto: any TrasportoFonti
) throws -> Fonte {
    let fonte = try trasporto.leggi(id: id)
    guard fonte.id == id else { throw ErroreFonte.identita }
    guard fonte.revisione == revisione else { throw ErroreFonte.revisione }
    return fonte
}
```

In Swift, `tramite` is the label used at the call site, while `trasporto` is the name used in the body. `try` marks a call that may produce an error. First we obtain a source; then we check that it is the requested one at the expected revision. The result is returned only after both checks.

If we entrusted checking only to the file transport, a second implementation could skip it. Here both checks belong to the core using the data. The protocol cannot stop an implementation returning `C` when we requested `B`; the core can, however, recognise the violation before accepting the result.

5.8 The change: from memory to a file

The first implementation, `TrasportoInMemoria`, looks up the identifier in a prepared map. The second, `TrasportoDaFile`, uses a map associating `A` and `B` with predefined JSON paths, reads the bytes and decodes them as `Fonte`. Both transports belong to the application; the core depends on their protocol rather than the concrete types.

The narrow change occurs where we assemble the program: we choose file reading instead of the in-memory map. The core and its contract remain identical. In the laboratory, both implementations are already present in the same executable, so we can compare them without compiling a second version of the check.

The positive case requires `A` and `B` at revision `r1`. We then prepare `B` with `r2`, leaving the text unchanged: refusal must depend on revision, without inferring that the contents necessarily differ. Another case returns a source with identity `C` when `B` is requested. Finally, we remove `B`. The curated file is absent in file transport; the map entry is absent in memory transport. The technical cause changes, while the application meaning of absence remains recognisable.

Scenario	Expected outcome for both transports	What we check
Current sources	Code 0 and identities A and B with r1.	The core accepts the two requested sources.
B at revision r2	Code 3, revision mismatch.	Changing the connection does not bypass revision checking.
C returned instead of B	Code 3, identity mismatch.	Protocol conformance alone is insufficient for accepting data.
B missing	Code 2, source absent.	Missing data do not become empty text treated as a successful read.

A positive outcome does not confirm the documents' meaning. Revisions are curated labels, as in chapter 4; they do not prove byte integrity. Quotations and summary completeness are not checked. The observable property is that both paths pass through the same acquisition check.

5.9 Repeat the component comparison

You need Swift 6 with Swift Package Manager and Python 3. From the repository root, run:

```
python3 quality-gates/storage/gate.py run -- \
python3 ai-agentica/esempi/velia/capitolo-05/esegui.py \
--verifica
python3 quality-gates/storage/gate.py check
```

The driver prepares a private copy of the manifest and sources, compiles once, and uses the executable for eight cases: four scenarios for two transports. For every case, it compares exit code and message with prepared criteria. In the positive case, it also checks printed identities and revisions. A compilation error prevents proceeding to the cases as though the modules were available.

Build files, caches and Swift manager configuration belong to the private cycle and are removed. Sources and curated fixtures remain in the repository, rather than the latest test's answers. No network, credentials, operational archive or Velia installation are required. This experiment verifies the educational package; it does not declare the real product built or qualified.

To observe only refusal of the outdated revision, choose `--trasporto file --scenario revisione-superata` instead of `--verifica`. The individual program returns 3, the

expected refusal. The collective driver instead returns 0 when every case meets expectations, including refusals. These answer different questions: what happens in this case, and whether the set of cases follows the criteria.

5.10 A variation that challenges the contract

Imagine replacing local files with a remote document service. We can reuse the objective: obtain an identified source and reject a different revision. We cannot conclude that the current protocol is sufficient for a connection that may keep waiting. The educational function is synchronous: it does not yet express asynchronous waiting, cancellation or a deadline. An adapter that fails to return control also prevents the core from examining its data.

The exercise is to describe the extension before writing a network call. Identify which information must cross the boundary, where to retain identity and revision checking, and how to distinguish a missing source, denied access and an unreachable service. Add a case where the person cancels while a read is in progress. Which component must receive cancellation, and what must happen to a response arriving later?

Then propose a simpler change: display each source on a separate line with its identifier and revision. Explain why this concerns application presentation and does not require moving revision checking into the transport. The supporting solutions discuss both variations without anticipating a remote implementation we have not yet built.

The criterion we can already apply is to look for the owner of a responsibility and the information required to verify it. Separating modules makes some dependencies visible; retaining a contract permits a checkable substitution. Understanding what happens when two operations overlap requires adding time to this map. That is the problem chapter 6 will begin with.

References for further reading

The collection's source guide identifies the real manifest, components and revisions. The educational package contains complete sources, inputs and commands. The chapter's fragments come from these two sources, kept distinct.

The Swift project's documentation on [protocols](#) and [access control](#) develops requirements, conformance and visibility between modules. Our example introduces only what is needed to examine the boundary; concurrency properties will arrive with the next chapter's problem.

6. Concurrency, the interface and state ownership

We have separated components and changed how a source is acquired without moving the core's checks. Now let the work take time. I ask Velia to compare two contracts in the conversation dedicated to that project. While waiting, I open another chat and prepare a note. The comparison's answer arrives when the note is already on screen. Where should it go?

The answer seems obvious: into the comparison's conversation. Yet the program need only use “the chat selected now” at delivery time to write in the wrong place. The service may have returned a correct result, and source checking may have succeeded, while the application still betrays the assignment.

This is a case prepared for the book, through which we will examine responsibilities present in the real sources. We are not reconstructing a documented Velia incident. We start with the structures and components introduced earlier, adding the Swift mechanisms needed to reason about waiting and updates. By the end, we must distinguish an answer still valid for a hidden chat from an answer superseded within the same chat. These are different situations requiring different decisions.

6.1 Looking elsewhere does not change the recipient

To understand the first problem, separate two questions: which conversation am I showing, and which conversation does the work belong to? Selection answers the first. The state owner answers the second. Here “owner” means the component responsible for retaining and modifying those data; it does not describe a person's rights or the language's automatic memory management.

At Velia's adopted revision, ChatWorkspace retains one controller per conversation. A controller is an object coordinating state and the operations through which the interface presents work. The `assistants` property associates a chat identifier with its `AssistantController`. This is the real declaration fragment:

```
private var assistants: [UUID: AssistantController] = [:]
```

Square brackets describe a dictionary: a UUID key maps to a controller. A UUID is an identifier used to distinguish entities; our reasoning depends on identity equality, rather than a temporal ordering of values. `[:]` creates an empty dictionary. `private`, already encountered at component boundaries, limits access to this property according to Swift's visibility rules.

When selection changes, the workspace finds the controller to display. Updates from controllers already created instead retain the original conversation's identity: the connections notifying content capture that identifier when constructed. They do not look for a new destination in the current selection whenever a response fragment arrives.

In the selection path examined in the sources, switching chats does not call the previous controller's response stop. Other activities are interrupted, and the state of the chat being left is saved, but we must not confuse these with cancelling the active response. This choice lets us return to the comparison and find work that has continued in the meantime. Figure 6.1 shows the distinction to retain.

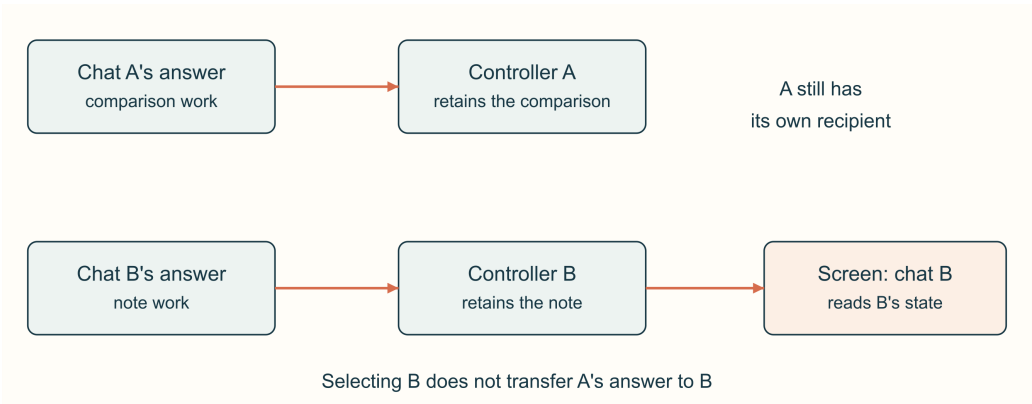


Figure 6.1 - Results reach their own chat's controller. Selection decides only which state is read on screen.

If we preferred a product that stops work when switching chats, we would need to introduce that rule explicitly and make it understandable to the person. It would not follow automatically from having only one visible window. We examine a concrete Velia solution; in the laboratory we retain its criterion: changing selection does not transfer a result to another owner.

6.2 Wait without occupying the whole path

The file read in chapter 5 finished before its caller continued. A remote response may take much longer. If we kept the path managing the interface occupied throughout the wait, the person might be unable to switch chats, press stop or even see an update.

In Swift, a function declared `async` can participate in asynchronous execution. In our case, it can begin a read, suspend while the result is unavailable, and resume when it becomes available. The caller writes `await` where it accepts that possible suspension.

Suspension concerns the current work: it does not require the program to stop every other activity.

“Possible” matters. `await` does not promise that the work will finish within a particular time or execute on a new thread. A thread is a program execution resource managed by the system. The concurrency mechanism organises work and waiting without requiring us to assign a thread manually to each request.

Concurrency and parallelism answer different questions. Two jobs are concurrent when their paths can make interleaved progress: while one waits, the other can proceed. They execute in parallel when they actually work during the same interval on different execution resources. Keeping a chat usable during a read first requires managing waiting well. Declaring a function `async` does not automatically turn a long synchronous calculation into work executed elsewhere.

A `Task` represents asynchronous work managed by Swift concurrency. It can await other operations and receive a cancellation request. It is not an agent: the existence of a task does not give it the ability to choose a plan. In `Task { ... }`, the braces contain the work to execute. Retaining the returned value allows us, for example, to request cancellation later with `cancel()` or await the result through `value`.

This chapter introduces the mechanisms for a precise problem. The primary reference is the Swift project's [concurrency chapter in *The Swift Programming Language*](#). It explains language suspension, tasks and isolation; it does not establish which conversation an answer in our application should belong to. That decision remains in the contract we are building.

6.3 Protecting access is insufficient to choose the result

In chapter 5, we encountered an actor in the store. An actor protects its isolated state by coordinating access through concurrency rules. For interface state, Swift also provides `MainActor`, a shared isolation domain suitable for updates that must occur on the application's main execution path. Writing `@MainActor` on a class places its isolated state and affected operations under that domain.

In the real code, both `ChatWorkspace` and `AssistantController` are annotated `@MainActor`. This helps prevent uncoordinated concurrent access to the same data. It does not establish that every arriving answer still belongs to the current attempt. To show the limit, I deliberately prepared an insufficient class in the laboratory. The following method is extracted from that class, also annotated `@MainActor`:

```
func acquisisci(_ id: String, tramite servizio: ServizioControllato) async {
    let risposta = await servizio.leggi(id)
    testo = risposta
}
```

`servizio.leggi(id)` awaits text. Afterwards, the method assigns the result to `testo`. It does not establish whether that result is still wanted. We start a first acquisition, then a second. We deliver the second answer and only then the first: the old text overwrites the new.

This error does not require two instructions to modify the same memory area simultaneously. Access can be properly isolated while the sequence is wrong for the intention. Protection against a *data race*, incompatible and uncoordinated concurrent memory access, does not replace checking a result's logical validity.

What can happen during `await`. A suspended isolated operation allows other admitted work on the same actor to proceed. When it resumes, state may have changed. This possibility is called *reentrancy*. The official proposal [SE-0306, *Actors*](#), by John McCall, Doug Gregor, Konrad Malawski and Chris Lattner, explains why isolation does not make an entire method containing waits indivisible. For us, the consequence is concrete: a condition checked before waiting may need checking again afterwards.

The synchronous section also needs attention. If we run a very long calculation inside a `MainActor` method, other activities requiring that domain must wait. In the laboratory, isolated sections compare a few identities and update a few properties. Heavy work needs a designed placement and appropriate transferable data; adding `async` alone does not solve that problem.

6.4 Identify the attempt that may still write

Return to the same chat. The comparison is running; I interrupt it and start another. The old result belongs to the right conversation but the wrong attempt. We therefore need an identity more precise than the chat identifier alone.

In the laboratory, we use a value carrying both references:

```
struct RiferimentoTurno: Equatable, Sendable {
    let chat: String
    let generazione: UUID
}
```

`chat` distinguishes the conversation; `generazione` distinguishes one of its attempts. The structure uses `let` properties, which cannot be reassigned after creating the value. `Equatable` permits equality comparison under the protocol's requirements. `Sendable` tells the type system that the value can cross concurrency boundaries under the

required checks; these value fields are compatible with that declaration. It does not make the reference valid forever or grant chat access.

When starting an attempt, the controller retains its current generation and returns a reference with the same identity. Before waiting, asynchronous work captures that value and the destination controller. When the answer arrives, it returns to that controller with that reference. This is the laboratory's acceptance method, in the `Conversazione` class annotated `@MainActor`:

```
func accetta(_ risposta: String, per riferimento: RiferimentoTurno) -> Bool {  
    guard riferimento.chat == id,  
          generazione == riferimento.generazione,  
          fase == .attesa else { return false }  
    testo = risposta  
    fase = .completa  
    return true  
}
```

Read it in decision order. `guard` requires every condition to be true; otherwise the method returns `false` without replacing the text. It checks the chat first, then generation, then whether the attempt is still waiting for an answer. Only then does it assign text and move the phase to `completa`, complete. `true` tells the caller that delivery was accepted.

There is no `await` between comparison and assignment. Checking and mutation occur in the same isolated synchronous section: another operation on that isolated state cannot intervene through a suspension in this method. If we inserted an asynchronous read after `guard`, we would need to reconsider conditions before the mutation. Protecting the controller makes that section dependable; choosing the conditions makes its decision correct.

At startup we retain the destination instead of consulting selection at the end. On replacement, we invalidate the old generation before creating the new one. Figure 6.2 follows the case where the previous answer arrives last: its content does not decide whether it may write; comparing its reference with current state does.

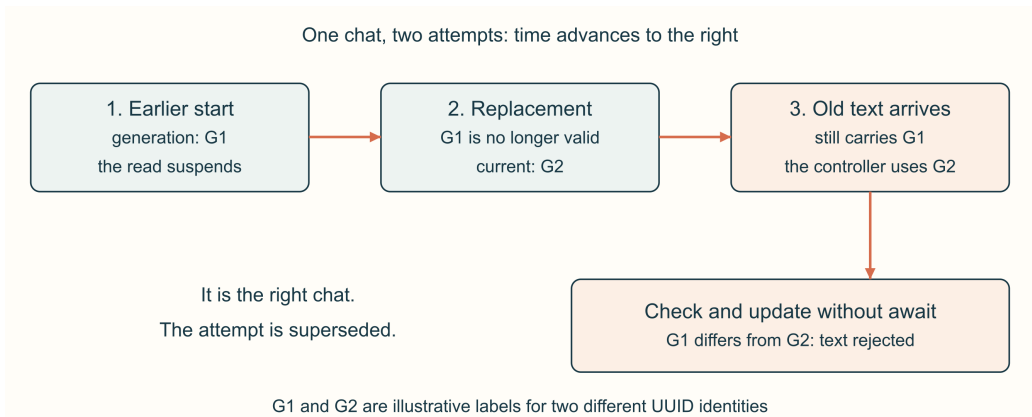


Figure 6.2 - The new attempt replaces the current generation. The previous answer returns to its own chat, but the check rejects its generation. G1 and G2 label two different identities.

What a generation identifies. Here the value is neither a model version nor a document revision. It identifies the attempt the controller accepts. If a document changes while the same attempt remains open, this comparison alone does not detect the change: the earlier chapters' revision checks remain necessary. Likewise, correct identity does not replace access verification.

6.5 Finding the criterion inside Velia

The laboratory reduces the problem to a final answer. Velia also receives intermediate updates. In the real `AssistantController.append` method, this condition appears before an update is used:

```
guard self.generation == generation, phase.isActive, !messages.isEmpty else { return }
```

`self.generation` is the controller's current generation; the `generation` parameter accompanies the received update. The check also requires an active phase and the presence of messages. For an admitted text update, the method then appends the fragment to the message under construction. The value captured when the response started lets us recognise updates belonging to previous work.

The final delivery path also checks cancellation and compares generation before applying the result. Error handling checks generation too: an old error must not put a new response into an error state. Following only the successful path would therefore leave a second door through which old work could change the chat.

The same need reappears in the store, with the core's own identities. `AgentRuntime` coordinates tool work; `AgentStore.finishStep` compares generation and the states of

run and step within the local transaction before admitting completion. The store does not delegate the entire decision to a caller's check made before a wait.

One distinction matters: that method can record actual resource consumption even after cancellation, before the check admitting the result as completion. Rejecting a superseded result does not mean pretending the work cost nothing or that no local data can change. We are protecting a particular transition and its result, while keeping other responsibilities separate. Nor does the store's transaction make an operation already performed on an external service reversible.

These steps are verifiable at the revision identified in the source guide. We examine them to connect the principle to the real product. This chapter's program does not compile Jarvis or certify that every Velia path correctly applies the same criterion.

6.6 Stopping is a local decision and a request to the work

Pressing stop expresses a decision: this attempt must no longer advance as the current attempt. Immediately interrupting every component involved is a further problem. An answer may already have reached the networking system, a library may fail to check cancellation, or an external operation may already have finished.

`Task.cancel()` signals cancellation to the task. The work must cooperate, for example by checking the request and terminating, or passing through operations that react to cancellation. `Task.checkCancellation()` lets it throw an error when the task is cancelled; reading `Task.isCancelled` lets it observe the signal. Neither undoes a write already performed elsewhere.

In the real `cancelResponse` method, Velia changes generation before calling `task?.cancel()`. The `?.` operator makes the call if the task reference exists. The method also starts the path asking the coordinator to stop the attempt and end folder access. These operations contain waits: local invalidation does not coincide with their completion. `stop()` does not await the cancellation task returned by the internal path.

In the laboratory, we preserve the essential order: `interrompi()` invalidates generation and sets the phase to `fermata`, `stopped`; whoever retained the task then calls `cancel()`. We prepared a service that deliberately ignores the signal and returns text anyway. The answer arrives, but `accetta` rejects it. This test specifically prevents us from attributing local state protection to the service's cooperation.

Why also study a service that does not cooperate? The official proposal [SE-0304, *Structured Concurrency*](#), by John McCall, Joe Groff, Doug Gregor and Konrad Malawski, describes cooperative cancellation. It helps explain why the signal and actual termination differ. Our controlled service forces an extreme late-delivery case; it is not a pattern to copy into production transport. A real adapter must release resources and propagate stopping as far as the service it communicates with permits.

This order transfers beyond chat: invalidate what may no longer write, request work cancellation, and handle any late return. If the task has already produced an external effect, we must also establish and communicate that effect. Merely rejecting text does not restore the world to its previous state.

6.7 Force the order that challenges the program

The complete laboratory requires Swift 6 and Python 3. It is a command-line program representing state observable by a possible interface, without opening a window or calling a model. Chats A and B are names prepared for this experiment, distinct from documents A and B in the previous comparisons.

A controlled service receives reads and keeps them suspended until the test case supplies the text. Before changing state, we await an explicit signal confirming that the read has started. We do not wait a few milliseconds hoping another activity has begun: on a slower computer, that assumption could be false.

A continuation retains where to resume. In the service, Swift's `withCheckedContinuation` mechanism suspends the operation and supplies a reference through which it can resume with a result. The service retains that reference; delivery removes it and resumes it exactly once. Forgetting it would leave work suspended; resuming it twice would violate the contract. The driver also imposes an external process deadline: a stuck case is a test error, rather than successful delivery.

The table identifies what to observe in the prepared cases. The two arrival-order variations for switching chat and replacing an attempt bring the total to eight executions.

Case	What it must make observable
Current answer	Its controller accepts text and completion.
Chat switch, in both arrival orders	A retains its result; B retains its own and remains visible.
Replaced attempt, in both orders	The old generation is rejected both before and after the new delivery.

Case	What it must make observable
Stop with a non-cooperating service	The task is cancelled; late text does not reopen the stopped chat.
Isolation only	The insufficient class overwrites new text with old: the counterexample's expected error.
Two independent reads	Both start; reversed delivery preserves result identities.

From the repository root, run every case:

```
python3 quality-gates/storage/gate.py run -- \
python3 ai-agentica/esempi/velia/capitolo-06/esegui.py \
--verifica
python3 quality-gates/storage/gate.py check
```

The outer command manages the test's transient space. The driver compiles with Swift 6 checks and treats warnings as errors; it then verifies text, identity, phase and completion of the expected waits. Sources and scenarios remain available; executables, caches and results are removed by the cycle.

In solo-isolamento, the criterion is to observe the intentional defect. Passing that case does not mean the class has been made correct: it means the experiment exposed its limit. In the protected cases, by contrast, the criterion is to prevent incorrect delivery. This distinction prevents us from reading one final success message as certification of every implementation present.

Checking these orders does not explore every possible interleaving of operations. Nor does it measure performance or the answer's semantic correctness. It makes some decisive sequences reproducible and verifies that the proposed solution respects its contract under those conditions.

6.8 Two reads that can progress separately

Let us transfer the reasoning to source collection. If reading the second document does not require the first document's content, we can start both before awaiting their combined result. In the laboratory, the dedicated method contains this fragment:

```
async let a = servizio.leggi("fonte-A")
async let b = servizio.leggi("fonte-B")
return await (a, b)
```

`async let` starts a child activity to obtain each value. The parent-child relationship means their lifetime is governed by the containing scope: the path does not abandon them as work without an owner. This is structured concurrency. The final line awaits

both results and returns them as a pair, preserving their association with `a` and `b` even if `B` finishes first.

The test waits until both reads have started, then delivers `B` before `A`. It checks that the pair's first item remains `A`'s content. It does not use a stopwatch to conclude that the program is twice as fast, or infer that the reads ran in parallel. When operations really are independent, this structure lets them wait without requiring one to finish before the other starts.

The pair is useful data to pass onwards: it contains distinct results without requiring two activities to modify one shared text freely. In real transport, we would also carry source identities and revisions, preserving the core's checks. If the second request instead depended on a decision based on the first document, independence would be false and we would need to await that decision first.

The `Task { ... }` creation used for the laboratory attempts differs from `async let`: it produces an unstructured task whose reference we explicitly retain and whose result we await in test cases. In an application, that reference needs clear responsibility for lifetime and cancellation. Child activities must nevertheless cooperate with stopping too: structure cannot force a service to terminate if it retains a suspended continuation without resuming it.

6.9 Try changing the situation

Imagine leaving two chats open while interrupting only the comparison in the first. Which data must change so the second answer remains valid? Before touching code, identify each result's owner, the generation to invalidate and the point where it must be checked. If one command cancels every available task, we have introduced a broader rule than the request received.

Another variation retains the attempt but changes a source before delivery. Explain why the generation may still match while the content must be rejected or qualified as a snapshot of an earlier revision. Identify the missing check and where it belongs. Attempt identity and document revision solve problems that meet within the same path without becoming equivalent.

Finally, replace the final text with a sequence of fragments. One more arrives after stopping. The check must apply to every update changing state, as well as to final delivery and errors. This situation makes the next step natural: connecting `Velia` to the model and distinguishing received data, an answer under construction and actual completion. Results must continue to belong to the right work even when they arrive one word at a time.

7. Connecting a model and reading a streamed response

A sentence appears in the comparison chat: “Delivery changes from ten to...” Then the connection breaks. Have we received an answer? We have certainly received data. We have also been able to show the person that work was progressing. We do not yet know whether the service completed the response, which elements are missing, or whether the visible text coincides with the final deliverable.

In chapter 6, we established who may modify a conversation and how to reject a superseded attempt's result. Now we must decide what is arriving at that owner. A byte sequence, a text fragment, a tool request and a completed response belong to the same path, but are not interchangeable.

We will follow Velia's real transport at the book's adopted revision. The laboratory will isolate request construction and reading a prepared stream: we can cut it at chosen points without credentials or paid calls. This will not test the network or model. It will let us understand a boundary before using it in the next chapter's loop.

7.1 Four choices behind “use this model”

The model name does not identify the whole connection. The provider is the organisation offering the service. The model identifies the processing capability requested from that service. The API interface establishes how to present inputs and read outputs. The transport implements the actual communication, with destination, authentication, timing and errors.

In the path we examine, `OpenAIChatModelTransport` defaults to the endpoint `https://api.openai.com/v1/responses`. An endpoint is the service destination receiving a particular kind of request. Velia builds a body for the Responses API and reads a streamed response. We cannot replace the provider name and assume another service accepts the same fields, returns the same events or supports the same capabilities.

Even two models from one provider may accept different parameter combinations. The name does not establish availability of images, tools, a particular reasoning level or the same context size. Velia's real configuration retains model and authentication mode; its paths check some incompatibilities and leave unknown identifiers for the service to validate. These are project choices at its revision, rather than an up-to-date catalogue of what every account may use.

Essential reference - The service contract. Official OpenAI documentation on [streaming responses](#) and [function calling](#), consulted for the Italian source on 28 October 2026, describes events and the exchange of tool requests. It helps us distinguish the public contract from client decisions. Velia's sources remain fixed at the source guide's revision: a new documentation page does not retrospectively update product code.

This distinction is practical. If a call fails, we need to understand whether the model is unavailable in the account, the body contains an incompatible parameter, authentication is invalid, or the connection fails to reach the service. “The model does not work” would merge problems needing different interventions.

7.2 The credential travels outside the context

To obtain the comparison, the model needs the assignment and relevant data. The credential instead lets the service recognise the client's access. The name token used for that secret does not mean the text units counted by the model here. Putting it into model instructions would confuse responsibilities and expose it unnecessarily.

The real transport method receives the body, credential token, authentication mode, deadline and an update function separately. This is the protocol requirement, reformatted for reading:

```
func complete(  
    body: Data, token: String,  
    mode: AuthenticationMode, deadline: Date,  
    onText: @escaping @Sendable (String) async -> Void  
) async throws -> Data
```

Data contains bytes owned by the value: on input it represents the request body; on output, the complete document returned by the path. `async throws` permits waiting and errors, already encountered in our Swift bridge. `onText` is a function the transport can call with a piece of text. Its `Void` result means it returns no useful value to the caller. The function may await, and `@Sendable` declares it suitable for use across concurrency boundaries.

`@escaping` indicates that the function may be retained and used beyond the receiving function's scope. It does not automatically assign an actor or decide whether text may still enter the chat. The update path must continue returning to the controller and generation checks studied in chapter 6.

In the API-key branch, the transport puts the token in the HTTP Authorization header with the Bearer scheme, and the JSON body in `httpBody`. A header accompanies communication; it is not a message in the model context. Client code

includes `KeychainTokenStore` and a system-backed storage implementation. Keychain is the macOS credential-storage service; access policies and biometric protection have their own paths. Retaining a secret in Keychain does not mean its value need never appear in program memory while used.

Good practice - Separate secrets from task data. OpenAI's [production best practices](#) require protecting keys and keeping them out of public code. In our path, we also distinguish body, headers and diagnostics: a connection error does not justify printing a credential. The laboratory receives none. The fact that its prepared request contains no secrets does not qualify all of Velia's credential management.

Velia also includes an access mode other than an API key. We do not reproduce its contract here as though it were universal authorisation available to every reader. We follow the branch needed to understand request and stream, keeping the described mode identifiable.

7.3 Construct the body and distinguish what it controls

The coordinator prepares instructions, available messages and descriptions of admitted tools. The real body contains the configured model, input, instructions, tools and the options `stream: true, store: false, parallel_tool_calls: false`. The laboratory constructs the same essential set without reproducing every product field.

`stream` requests a streamed response. `store: false` concerns storing the response for later retrieval through the API, as described in the [conversation state guide](#); it does not prove that the service retains nothing, and does not govern Velia's local archive. `parallel_tool_calls: false` expresses a choice about emitting tool requests. The real path's coordinator also checks that output contains no more than one call. The body value and result check have complementary roles.

The transport creates a `POST` request, identifies the body type as `application/json` and requests `text/event-stream` in response. It uses an ephemeral session without a URL cache or cookie store in the examined configuration. The `OpenAIRedirectPolicy` delegate refuses redirects: it does not automatically follow another destination. These are facts readable in code; verifying that every distribution preserves them requires appropriate client and network tests.

A favourable HTTP response indicates that the request reached a path returning that kind of outcome. It does not establish that the body contains the expected deliverable. The real method checks HTTP status and content type before reading events. For an HTTP error, the program collects a bounded portion of the error body and translates it into its diagnostic path.

We therefore have at least three distinct checks: communication validity, provider response validity and result admissibility for the application. Figure 7.1 places them in sequence. The boundary between the latter two will return in chapter 8, when a technically completed response proposes a function to execute.

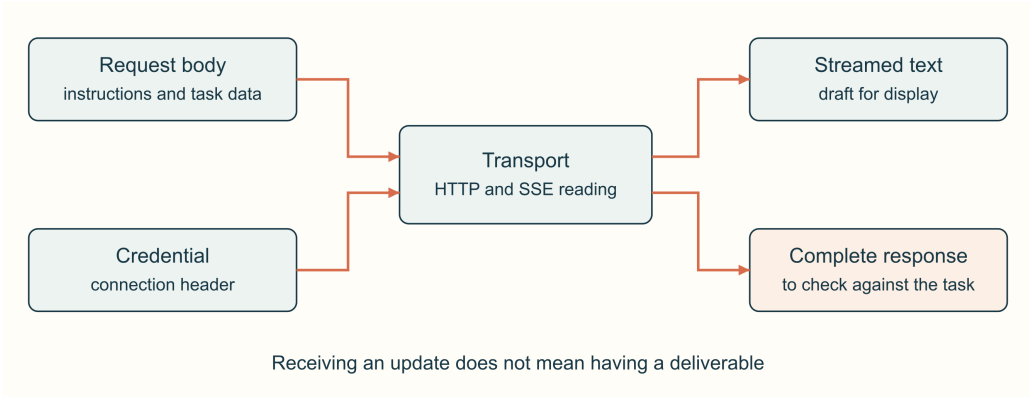


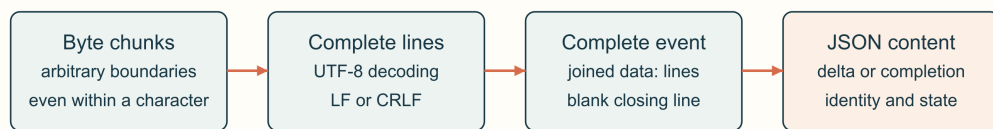
Figure 7.1 - The credential serves the connection. Task data enter the body. Receiving text and admitting a complete response are separate steps.

7.4 From bytes to an event, from an event to text

The service can send a response using Server-Sent Events, abbreviated SSE: a text format delimiting events within a stream. In the subset we need, data: lines contain an event's content and a blank line ends its block. The content is then interpreted as JSON.

A chunk received from the connection need not coincide with a line or event. It may contain several events, half a line, or only some bytes encoding a character. UTF-8, the stream's encoding, can represent a character with several bytes. Converting each isolated packet into a complete string can therefore damage valid text.

The laboratory retains bytes until it finds the line ending; only then does it decode them. It then accumulates data lines until the event ends. Its `LettoreSSE` retains these data between one `ricevi` call and the next. Figure 7.2 shows why two reconstructions are needed: first lines, then the event.



The reader retains incomplete parts between deliveries

End of bytes without completion: incomplete response

Figure 7.2 - Received chunk boundaries do not establish content boundaries. The reader retains parts that are still insufficient without promoting them into an answer.

Further explanation - Which part of SSE are we building? The **WHATWG HTML standard** defines event encoding and interpretation. Our educational reader handles LF or CRLF lines and multiline data; it does not implement the full browser EventSource interface, reconnection, resumption identifiers or every format variation. These limits prevent us from presenting it as a universal SSE parser. Service rules must be added above the general format.

In the Responses protocol, the JSON type field describes what happened. `response.output_text.delta` carries part of a text, called a delta. `response.completed` concerns response completion. Other events describe errors, incompleteness, refusal or progress of different items. The field name type does not make checking its existence sufficient: its contents must correspond to the expected event.

Function-argument deltas deserve particular care. A fragment such as `{"id":` is not a usable request. The laboratory executes nothing when it receives such deltas; it makes the call available only when reading complete output. Later, we can study different optimisations, but they must preserve a verifiable condition preventing action on incomplete arguments.

Now read the first two events of the fixture representing an interruption. Blank lines belong to the format and separate the blocks:

```
data: {"type":"response.created","response":{"id":"r-1"}}
```

```
data: {"type":"response.output_text.delta","delta":"È una bozza. "}
```

The first block announces creation of response `r-1`. The second carries a text delta, “È una bozza.”, meaning “It is a draft.” The blank line ends the update event, rather than the entire response: a complete event does not mean the response-completion event has arrived. In the fixture, the stream ends exactly here, without that confirmation.

Before running the case, we can predict two different outcomes. The update function will receive text and can display it as a draft. The path waiting for a complete response must instead report interruption. If the program returned the draft as a successful result, we would have located the error in content promotion without debating the sentence's plausibility.

The function variation changes event content but retains the criterion. An event completing a piece of arguments does not necessarily complete the response containing them. In this increment, we await final output before constructing a *Chiamata*, a call. Chapter 8's loop will therefore receive a complete proposal whose capability, identity and access it must still check.

7.5 Provider completion and the draft on screen

In the real `ChatAgentStreamParser`, the completion event requires status completed, reads final items, compares response identity and reconciles items received during streaming with the final ones. Conflicting or still-incomplete items meet specific refusals. The coordinator then receives that complete document.

The laboratory adopts a smaller contract. It first requires response identity, then completion with the same identity, completed status and output from the admitted subset. It accepts text messages, complete calls and reasoning data retained without interpretation. It does not reproduce Velia's parser registry of item indices. The decisive check at the end of the bytes is short:

```
func termina() throws {  
    guard concluso else { throw ErroreFlusso.incompleta }  
}
```

The end of the stream does not set `concluso`. Processing a valid final event does. If the connection ends after a plausible sentence but before that event, `termina` reports incompleteness. It does not estimate how likely it is that only a few words were missing.

In the positive fixture, deltas compose “È una bozza.”, “It is a draft”, while the final text is “È il testo finale.”, “It is the final text.” The difference is intentional: the consumer must use the complete deliverable as the authoritative value for this path, rather than assume concatenation of what appeared is sufficient. In the real product, we observed a `resetText` update before presentation of the final text processed by the coordinator. Corrections or qualifications of the deliverable must also reach the right chat.

Key concept - Opaque data retain a function without becoming an explanation. Some responses include reasoning items that may contain encrypted data to return in the continuation of the conversation. The laboratory retains the serialised object without trying to decrypt it. Its presence is not a verified transcript of the choice's internal causes and must not become scientific evidence of the model's reasoning. Retaining protocol-useful data and interpreting behaviour are different jobs.

A refusal must also be distinguished from an ordinary task result. The laboratory signals it separately and does not admit it as the comparison note. The real code has a broader presentation policy: the parser can forward refusal deltas, and `outputText` includes that content too. We do not attribute our experiment's restrictive policy to the product. In both cases, a refusal message must not be treated as evidence that the comparison was performed.

7.6 Deadline, cancellation and reconnection are not synonyms

The deadline establishes until when work is admitted; cancellation expresses a stopping request; connection closure is a transport fact. The service may already have processed part of the request before any of these events. A new submission is therefore not automatically a continuation of the previous one.

The real transport configures session timing, checks cancellation and deadline while reading bytes, and invalidates the session in its cancellation handler. The laboratory checks these conditions between prepared chunks. It does not model a connection remaining silent for minutes: that case needs testing of the network layer and its timeouts. A check written inside the byte loop cannot execute until a byte arrives and the loop advances.

The educational reader also imposes a total limit of 524,288 bytes per stream, chosen to bound this experiment. This is neither the model's context size nor a documented service limit. A source counter alone would not constitute a limit: a comparison must refuse an overrun. The dedicated test lowers the maximum to 32 bytes and checks that refusal.

If the stream breaks, we can retain a draft on screen marked as interrupted, or remove it according to the presentation contract. We cannot silently promote it into a completed answer. If we want to retry, we must establish which inputs to resend, which identity to assign to the new attempt, and how to treat operations already performed. In the next chapter, this problem will become clearer because the response may have requested a tool.

7.7 Repeat the cases without calling a service

The chapter 7 laboratory contains the reader, request builder and prepared transport. The SSE fixtures are curated sources written for the book, rather than account recordings. The transport delivers their bytes in chosen chunks and calls the function collecting the draft. It requires Swift 6 and Python 3.

From the repository root:

```
python3 quality-gates/storage/gate.py run -- \
  python3 ai-agentica/esempi/velia/capitolo-07/esegui.py \
  --verifica
python3 quality-gates/storage/gate.py check
```

The thirteen cases cover one-byte and seventeen-byte chunks, CRLF lines, multiline data, a complete call following fragmented arguments, truncation, provider error, incomplete response, identity change, malformed JSON, refusal, expired deadline and byte limit. Delivering one byte at a time also crosses the interior of the accented character: final text must remain identical to delivery in larger chunks.

The check distinguishes draft from final text, retains opaque data and verifies that argument deltas have not become executable calls. The program opens no HTTP connection. These cases qualify the reader and educational contract for the prepared cases without establishing authentication, account availability, proxy behaviour or a real model's quality.

A later connection test must separately check destination, headers without exposing secrets, content type, HTTP errors and stopping during a network wait. Model evaluation must instead compare requests, sources, answers and criteria on cases showing variability. Separating questions lets us understand what to correct when a test fails.

7.8 A variation to take into the loop

Imagine the connection ending after displaying the comparison but before completion. The person asks to save that text as a document. Describe what state would permit proposing the save of an interrupted draft without declaring it a verified comparison. Consent to save concerns the document we actually have; it does not establish that the service had finished.

Then change the stream: a request to read B arrives, followed by provider completion. Identify which checks remain before the operation and which data must return to the next request. Having a complete call solves the protocol-reading problem; it does not yet establish access rights, source revision or task outcome.

This is the chapter's destination. We have a way to distinguish updates from a response admitted by the transport. We can now connect a complete proposal to a tool, return its outcome to the model and decide when to stop the whole job.

8. Velia's first executable loop

The transport returns a completed response. Inside it, we find a request: read source A at revision r1. We do not yet have the comparison. We have an operation proposal from which new data may arise. The step making this path agentic will be returning that outcome to the model and letting it choose what follows.

If we always read A, then B, then called the model to write the note, we would build the workflow discussed in chapter 2. Here, the loop contains no mandatory list of those reads. It receives a structured proposal, checks it, executes an admitted capability and prepares a new request containing what happened. The difference concerns who chooses the next step; the program continues to own access rights, state and stopping conditions.

Velia implements this alternation in the chat coordinator and uses the core to execute and record steps. We will follow those boundaries, then build a complete educational increment executable with the Swift tools already used. Prepared sequences will replace the model: we can verify the program and provoke its errors without confusing this test with evidence of the model's ability to choose well.

8.1 Shape the work before the loop

We retain the assignment: compare delivery times in sources A and B at revision r1, and support the note with retrievable passages. Support and the initial comparison's other contents remain outside this increment's scope. An inaccessible source must become a declared limit rather than be invented. The laboratory resources are two small in-memory texts serving the same purpose as the earlier chapters' curated data.

Before choosing classes, distinguish three information states. A source may not yet have been requested; it may have been acquired; or its read may have produced denied access. The last condition is not equivalent to the first. If we have never tried reading B, we have no observed outcome on which to base “access to B is denied”.

In the laboratory, an observation retains an identifier, revision, status and optional text. “Observation” here means the outcome received from a program capability. It does not give that outcome more read authority than its origin warrants: the prepared catalogue can attest to its own read, rather than the truth of the commercial contract represented by the text.

A final response must also identify what is missing. “Partial”, without a recognisable gap, would be an unhelpful label. We must connect the absence to a requested source

and a work outcome. This choice establishes the completion criterion before the message announcing that the work is finished.

8.2 Where the alternation lives in the real project

In `ChatAgentCoordinator.respondWithMemory`, we find a loop preparing capabilities and instructions, sending a request and reading items from complete output. The model call itself passes through the execution path as the `chat_model` capability, with storage and consumption accounting. It is not exempt from task limits merely because it does not modify a file.

The real step distinguishing function requests is:

```
let calls = outputs.filter { $0["type"] as? String == "function_call" }  
guard calls.count <= 1 else { throw ChatAgentError.invalidResponse }
```

`filter` selects structured output items: the program recognises the protocol's expected type rather than searching the user's sentence for “read”. The second check refuses more than one function request in that output. This is the examined coordinator's policy, consistent with its request option; it is not a property of every agentic system or model.

The coordinator retains complete received items, including opaque ones, to construct the continuation. If a function is present, it checks name, call identifier and arguments, constructs the executable capability and uses `AgentRuntime`. The result becomes a `function_call_output` item associated with the proposal's `call_id`. The next iteration therefore receives another fact: what the tool returned.

If no function is present, the delivery path applies other criteria and chooses the run's transition. The real code distinguishes the provider's `completed` status from application states `completed` or `partial`. The latter also depend on steps, operations and evidence. Provider completion closes a response; assignment completion requires broader conditions.

Key concept - A loop does not belong to one class by definition. The `VeliaCore` target contains the tool runtime and store; at the studied revision, the chat coordinator belongs to the desktop. Calling both an “agentic core” in general discussion does not turn them into one module. Chapter 5's map also helps us find this concrete distribution.

8.3 Proposal, available capability and permitted operation

The model may propose `leggi_fonte` because the program described that capability to it. The description includes a name, purpose and argument schema. In the laboratory, only identifiers A or B and revision `r1` are admitted. The schema requires both fields and excludes additional properties. The executable catalogue nevertheless retains its own checks: receiving a proposal conforming to the model's expectations is not equivalent to having permission.

A tool is therefore more than a name in a request. It needs a concrete function, readable arguments, an admitted resource, an outcome and error behaviour. A skill could teach how to compare sources and handle exceptions, but could not grant access to B. We do not yet load a skill package here: we retain the distinction needed in the dedicated chapters.

In our `CatalogoFonti`, reading is admitted only after name and field checks. This fragment shows the boundary before content access:

```
guard chiamata.nome == "leggi_fonte" else { throw ErroreCiclo.strumento }
guard let dati = chiamata.argomenti.data(using: .utf8), dati.count <= 4096,
    let campi = try JSONSerialization.jsonObject(with: dati) as? [String: String],
    Set(campi.keys) == ["id", "revisione"],
    let id = campi["id"], let testo = fonti[id],
    campi["revisione"] == "r1" else { throw ErroreCiclo.argomenti }
```

The first guard prevents another name selecting an arbitrary function. The second imposes size, exact argument structure, catalogue membership and the required revision. A field such as `percorso`, meaning path, is refused even if it contains a plausible destination: the catalogue is not a reader for any file on the computer. Set constructs a set, used here to compare field names independently of their order in JSON.

Only after those checks does the catalogue record a read attempt and apply the prepared access rule. If B is denied, it returns an observation without text and with status `accesso_negato`. The caller cannot bypass that result by changing instruction text. This protects the laboratory catalogue; a real resource requires checks in the component actually owning the data.

In the product, `AgentRuntime` resolves the name among registered capabilities and checks the input schema before asking the store to start the step. `ChatNativeTool` also participates in access checking within its own path. Reading these points prevents us from confusing the description sent to the provider with authority to execute the operation.

8.4 Return what happened to the model

The first prepared response proposes reading A. The catalogue returns its text with identity and revision. Before the second call, the program retains both the complete model proposal and the observation. The order establishes a relationship: this is the request, and this is its result.

`call_id` identifies the particular call, rather than the conversation or document. Two reads requesting A may have different call identifiers. If the same identifier is repeated within this loop, we treat it as an error before executing again. This avoids an unintended local repetition; it does not by itself implement durable idempotency for a remote action.

In the laboratory, recording the result in the next context uses this form:

```
elementi.append([
    "type": "function_call_output",
    "call_id": chiamata.id, "output": risultato
])
```

`risultato` is the serialised observation. If B is denied, it contains denial, rather than an empty document or imagined text. The second or third request can therefore receive information that changes what follows. Figure 8.1 highlights the return path distinguishing this construction from a single model call.

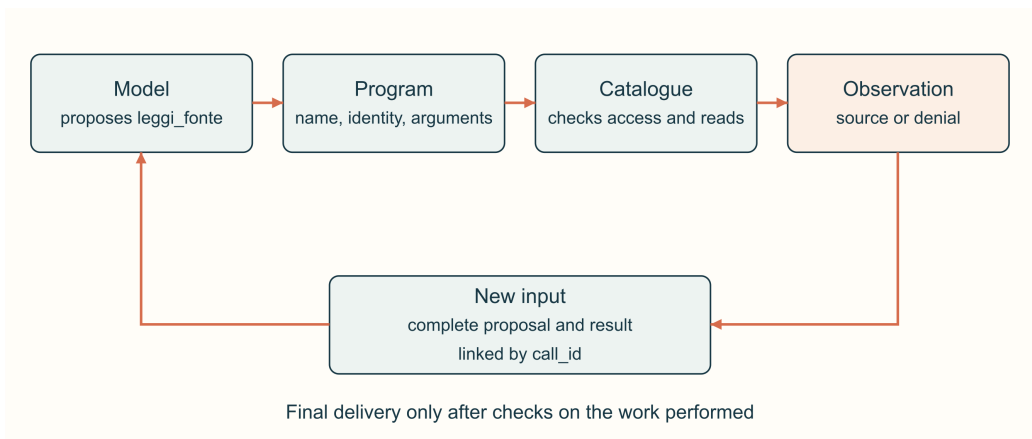


Figure 8.1 - The tool outcome enters the new request with its call reference. The model can propose what follows; the program retains authority to admit it.

Essential reference - Continuing a function call. OpenAI's [function calling guide](#) distinguishes the model proposal from application execution and describes returning through `call_id`. Here we apply that contract to controlled local reads. Catalogue results do not show that a real model would choose the same reads or respond correctly to denial: that is a model-evaluation question.

The code compares names and fields because they belong to a protocol. It does not classify linguistic intentions. “If a read is denied, declare the limit” instead belongs to instructions for whoever proposes what follows. In the laboratory, that decision is prepared in a fixture; with a real model, we must also test it on requests formulated in different ways.

8.5 Govern work during waits

The loop awaits the transport and tool. At those points, the person may have stopped the attempt. We therefore reuse chapter 6's criterion: an actor owns phase, generation and observations, while the work retains its startup reference.

In the laboratory, `registra` is synchronous with respect to actor isolation:

```
func registra(_ dato: Osservazione, per riferimento: UUID) throws {
    try controlla(riferimento)
    osservazioni[dato.fonte] = dato
}
```

`controlla` requires generation and phase still to be admitted. There is no wait between checking and recording. The loop also checks after the model returns, before accepting its proposal. Task cancellation and state invalidation remain different mechanisms; a new activity must not be able to use the stopped activity's reference.

This actor retains only in-memory data. Closing the process loses them. Velia's real store has persistence responsibilities this increment does not reconstruct: we do not present an isolation-protected method as recovery after a computer stops. Today's work is to make the alternation understandable while preserving boundaries already known.

Even a proposal that is always formally valid may fail to reach a conclusion. For example, the model could request A without ever acquiring B. In the laboratory, we permit at most three model calls: two reading opportunities and one delivery opportunity in the prepared path. The loop does not enforce that use; after a third turn still lacking a deliverable, it stops with a limit.

Three is an experimental parameter, rather than a general recommendation for Velia. We do not describe the product's `while true` as unbounded merely because it lacks

this same counter. In the real path, runtime and store also govern deadlines, resources and consumption. Actual conditions must be reconstructed through the components applying them.

8.6 When a response can become a deliverable

The third prepared response contains no function. Its message contains a JSON note with declared status, text, quotations and missing sources. The program does not take “complete” as evidence. It looks for observations that should support it.

The educational contract requires an outcome for both sources. From those outcomes, it derives which were denied. A complete note is admissible only if no required source is missing; a partial note must identify exactly those missing. For each quotation, the check requires an actually acquired source, the same revision and an excerpt present in the read text. Every acquired source must have a reference in this small comparison's deliverable.

Figure 8.2 shows both paths. Following denial of B, a partial note with the corresponding gap can be useful. A note claiming both documents were compared is instead refused. Application state derives from work and contract, rather than an optimistic sentence.

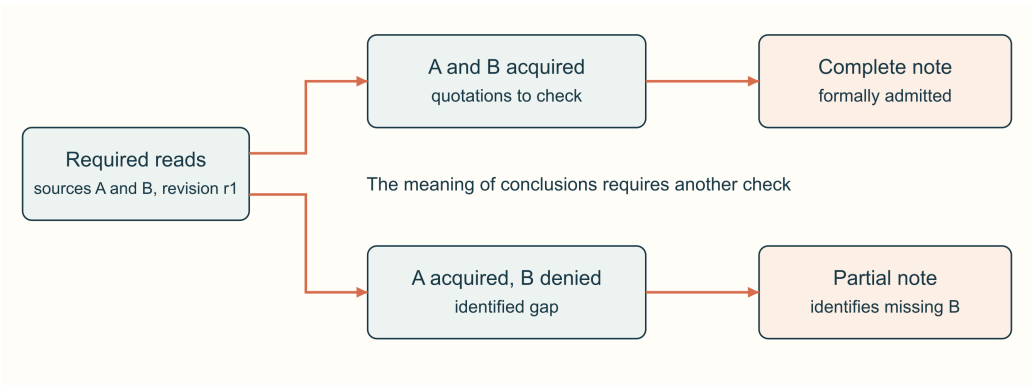


Figure 8.2 - Completing the provider response does not mean completing the comparison. Available observations determine which deliverables can be admitted.

Chapter 4's limit remains. An accurate excerpt can accompany a wrong conclusion. The prepared *significato-non-valutato* case quotes the ten-day and fifteen-day sentences but declares that the deadline decreases to five. Reference checking can admit it. It has no rule for understanding the sentence's meaning or verifying that every commercial condition was preserved.

Calling the final phase “validation” does not fill that gap. Evaluating meaning requires a check appropriate to the question and cases capable of disproving it. A human reviewer or another model might intervene, but the name “reviewer” would not establish reliability. We would also need to test that judgement on correct, incorrect, ambiguous and incomplete conclusions.

Further explanation - Four meanings of “finished”. The connection may have stopped sending bytes; the provider may have completed the response; the tool may have finished an operation; the assignment may have met its criteria. These events can occur at different times. In the laboratory, we chiefly distinguish a completed response from a note admissible under formal checks. We do not infer content understanding or the person's complete satisfaction from those checks.

8.7 Read the whole loop in the accompanying sources

The chapter 8 program reuses chapter 7's `Flusso.swift` without copying it. `PrimoCiclo` retains conversation items and the set of call identifiers already encountered. At each turn, it constructs the input, awaits a complete response, verifies that work is still active, and distinguishes a deliverable from a tool request.

The deliverable passes to state checking. The tool request passes through the catalogue and returns as an observation. A structural error interrupts the laboratory; a recognised but denied read instead produces an outcome for the continuation. This is the experiment's explicit policy. `Velia`'s real code can turn various tool errors into unavailable observations while retaining a distinct path for stopping and other failures.

Laboratory verification also checks the next input: after reading `A`, its `function_call_output`, reference `c-A` and opaque items from the previous response must be present. Following denial of `B`, the next turn must receive `accesso_negato` with reference `c-B`. Checking only the final printed text would not establish this return path.

From the repository root:

```
python3 quality-gates/storage/gate.py run -- \
  python3 ai-agentica/esempi/velia/capitolo-08/esegui.py \
  --verifica
python3 quality-gates/storage/gate.py check
```

The nine sequences have different criteria, grouped in the table. Read counts are observed on the catalogue, so a late refusal cannot conceal an operation already executed outside the contract.

Prepared situation	Criterion to verify
Complete comparison	Two reads, observations in the continuation, note admitted as complete.
B read denied	The next turn receives denial; the note identifies B as missing and remains partial.
Invented completeness	B is denied but the text declares complete: deliverable refused.
Premature delivery	The model proposes the note without acquisition: no reads, deliverable refused.
Unknown tool or extraneous arguments	No reads before refusal; these are two separate sequences.
Duplicate identifier	The second proposal does not cause a second read.
Turn limit	Three admitted proposals do not cause a fourth model call.
Meaning not evaluated	Valid references and a wrong sentence expose the formal check's limit.

Fixtures prepare the choices; the program genuinely implements waiting, call recognition, local reading, recording and new input. The supported conclusion is therefore narrow: the loop can be exercised and checked with these decisions. A real model's ability to make them remains to be evaluated, just as the full product requires its own verification.

8.8 Transfer the loop without transferring assumptions

Add source C containing an exception to delivery terms. Before changing the tool schema, decide whether C is mandatory from the start or should be identified after a read. In the first case, the delivery contract must include it. In the second, we need a rule for acquiring a reference emerging from the work while keeping access and identity checkable. Merely increasing the maximum turns does not settle that choice.

Then consider another action: saving the note. The function result must let us establish which file and revision were written. If the connection breaks after saving, recreating the in-memory loop does not make repetition harmless. Local protection against duplicate identifiers does not survive a restart or replace confirmation from the external resource. Changing the action requires new tests, even though the loop's general pattern remains useful.

Finally, stop work during a read. Identify who invalidates the attempt, who requests task cancellation, and which component must reject late data. Distinguish what we can prevent in delivery from what we may need to establish outside the process. The supporting material's explained solution proposes a criterion without claiming remote saving has already been built.

We have now connected the first bricks: an assignment with criteria, components with distinct responsibilities, a wait retaining ownership, a completed response and a loop using outcomes. The next chapter will enter the executable capability itself: how a function becomes a tool with arguments, results and errors precise enough to use and check.

Supporting material - Solutions, glossary and further reading

The following solutions explain their reasoning: they show criteria and limits without imposing one architecture where the problem permits alternatives. The activities come from chapters 1-8. To use them as exercises, formulate your own answer first, then compare what each solution makes observable.

Save the comparison as a PDF

In chapter 1, the result changes from conversation text to a saved document. Ready content is an initial condition, but we must also know which source revision it describes. Conversion produces a particular PDF revision; saving must concern those bytes and an admitted destination. A model message announcing a file does not replace confirmation of the operation.

If the folder is not writable, the deliverable can retain the text and identify the problem, or ask for a permitted destination. It must not declare the file saved. If a source changes after conversion, we must decide whether the PDF remains an identified snapshot of the earlier revision or must be regenerated to meet the current assignment. The decision depends on the criterion received and must be explicit.

A useful test observes content, revision, destination and effect. One variation interrupts the program after writing but before confirmation: repeating the save without knowing what happened is not a general solution. We may need to check the expected file or reconstruct the operation with an identifier.

Choose between a call, workflow and agent

For the isolated call, we can use two prepared excerpts and a request to explain the difference. Acquisition occurred outside the call and checking evaluates the comparison's fidelity. If a source is missing, the solution must not invent it.

For the workflow, we can use periodic checking of stably structured files: acquire both, extract five known fields, verify availability and ask the model for a note. The procedure may branch for errors and missing data. Its suitability comes from path stability and defined criteria, rather than the absence of a model.

For the agentic loop, we can use document research where discrepancies require different sources. The model proposes what follows after reads, while the program

retains access, identity and stopping conditions. One possible limit is a missing mandatory source; the expected result may be a question or partial deliverable. Flexibility must be compared with a simpler procedure on the same cases.

More candidates for resumption

Chapter 3's extension first requires a scope contract. The program can make several admitted candidates from the conversation or authorised project available, retaining identities, states and original-request references. Candidate number and type must be justified by capability and access, rather than chosen by searching the user's words.

The model evaluates metadata relevance to the current request. If two assignments remain compatible with what the person said, a question is a valid outcome. A recovery request must specify an exact identity, and the program must recheck admissibility and access at reading time. The result remains a set of past references: it imports no permissions and re-executes no actions.

A program test can use a prepared choice containing another conversation's identifier and require refusal. Model evaluation can use paraphrased or ambiguous requests and judge the choice or clarification. The two test families answer different questions.

A third source and insufficient verification

For chapter 4's attachment C, add an expected identity and an available source with a revision. A statement about support may cite C. The program can check that the source exists, its revision matches and the excerpt is present. If C is missing, the expected source remains absent.

The check may nevertheless succeed if the text omits a decisive exception in another paragraph. An additional check must receive the question, statement, cited passage and relevant context, then verify whether conditions and exceptions change the conclusion. A model judgement still requires evaluation through curated cases and comparisons; the word “reviewer” does not automatically make its outcome reliable.

Another extension concerns version identity. Fixture labels do not prove byte integrity. Connecting revision and content requires defining who assigns identity and how a hash is associated with the acquired source. A hash can detect a content difference, but does not alone establish authorship or the document's truth.

Change the transport and preserve responsibilities

In chapter 5, transport returns a source while the core verifies that identity and revision match the request. Preserve this division when moving to remote reading. The result must identify the source; an error must distinguish at least absence, denied access and inability to reach the service. Transport can translate technical responses into those outcomes without turning a failed connection into an empty document.

The laboratory's synchronous protocol does not express work over time. Before networking, decide how to wait without blocking the interface, communicate a deadline and request cancellation. The coordinator retains the identity of the assignment using the read; transport must react to cancellation as far as the connection permits. A late answer must not automatically update the new conversation or another assignment. Identity and revision checks remain necessary even after a technically successful connection.

One useful test forces a conversation switch during the wait and observes where the result goes. Another returns revision r_2 although r_1 was requested: the new transport must still meet the core's refusal. The local laboratory does not perform those network reads or temporal tests; these are extension-design criteria requiring the next chapter's concepts.

The presentation variation has another owner. To display each source on a distinct line, change message composition in the CLI using the already accepted source data. There is no need to move checking into transport or change the protocol. The proposed version's message already contains identity and revision: separating these into two lines, one per document, reveals the difference between a visual choice and a data-admissibility condition. The presentation check's expected message must follow the new format too; refusal outcomes remain unchanged.

Interrupt one chat and let the other continue

In chapter 6's variation, screen selection is insufficient to identify the work to stop. Retain chat A's controller and its attempt reference. Invalidate only that generation and request cancellation only for the corresponding task. Controller B retains its current identity: its answer remains admissible if it passes chat, generation and phase checks.

A transferable test starts both jobs, interrupts A and delivers both answers anyway, in both orders. A must reject its late text and remain stopped; B must complete with its own text. Changing selection during the test must not change these outcomes.

Selection and stopping are different commands, even when one window provides both.

If a source instead changes within the same attempt, generation can remain identical. Transport returns actual identity and revision; the core compares them with those requested. When the assignment requires the current revision, a result based on an old source must be rejected or acquired again. If the mandate permits a historical snapshot, the deliverable must make that condition recognisable. The attempt reference alone does not settle the choice.

For a streamed answer, checking must accompany every fragment capable of changing text or phase. Final delivery and a late error must also refer to the still-admitted attempt. Checking only the first fragment lets later ones pass through changed state. Within isolated code, comparison and update must remain in one section without suspension; another wait requires reconsidering conditions.

The independent-read variation does not justify one mutable text shared among all jobs. Each read can return an identified value; the coordinator composes values and the core checks sources and revisions. If read B depends on A's content, resolve that dependency first. Concurrency is appropriate when the relationship between operations permits it.

Save an interrupted draft and continue a call

Chapter 7's variation first requires recognisable content state. The draft received before interruption has not met transport completion. We can propose saving it as an interrupted draft if the person wants, retaining that indication in document and outcome. We cannot attribute comparison verification to it or assume the absent text irrelevant.

Saving then requires operation checks: exact content, admitted destination and confirmation of the write. The person's request may authorise saving, but does not retrospectively turn the incomplete response into provider completion. If the document must meet complete-comparison criteria, resolve the gap before declaring successful delivery.

A read call received within a complete response instead opens the tool boundary. Name, identifier, arguments, resource, revision and access must be checked. Its result must return with the particular call reference. Fragments displayed on screen do not replace the complete proposal, and provider completion does not establish execution of the read.

A third source and an effect to establish

In chapter 8, variation C changes the work contract. If C is mandatory from the outset, add it to required sources and delivery criteria. If it emerges from B's content, the program must retain the proposed reference and resolve it within an admitted scope, without allowing a document to grant access. The model may choose to investigate; the component owning the resource continues to check reading. The turn maximum must allow the new path, but does not itself decide which sources are necessary.

When moving to saving, the tool result must identify written content, destination and available authoritative confirmation. `call_id` relates proposal and observation in the dialogue; an in-memory identifier set does not prevent repetition after restart. We need the resource's idempotency contract or a way to reconstruct the effect. If we do not know whether a file was already written, delivery must represent that uncertainty before retrying.

For stopping during a read, the state owner invalidates the attempt reference, whoever retains the task requests cancellation, and the store rejects recording against the old generation. An effect already produced outside the process still requires separate investigation. In the in-memory reading laboratory, we can observe rejection of delivery; we cannot infer an ability to undo external writes.

A transferable test must also check the continuation. After a denied read, model input must contain denial and call reference; an incompatible complete note must meet program refusal. Choosing a partial note instead needs evaluation on the real model, rather than being treated as established by fixtures preparing it.

Collection glossary

Agent. A unit entrusted with a task and the possibility of choosing how to proceed. In this journey, the model participates in that choice. The term describes a software responsibility and does not imply consciousness.

Agentic application. A system connecting decisions, state, tools, checks and outcomes around an objective. It may contain one agent or distinct participants.

Context. Material received by the model in a particular request. It may include instructions, messages and references; it does not coincide with everything the application retains.

Task state. The program's representation of work: objective, activities, acquired data, errors, waits and outcomes to verify.

Run. A work attempt identified by the system. A conversation can contain several; resumption must recognise which data and actions belong to each.

Tool. An executable capability with defined arguments, results and errors. A model request must be checked before producing an operation.

Skill. Instructions and resources for tackling a task. It can guide method and tool use; it does not independently create new access rights.

Workflow. A process with activities, conditions and branches defined by the program. It may use models and await events.

Mandate. The received objective with conditions, resources and permitted actions. It is kept separate from remembered preferences.

Adapter. A component connecting an application contract to a concrete service or resource, translating requests and outcomes.

Swift package, target and product. The package groups sources and manifest. A target is a build unit with its own dependencies; in the Swift path, its code forms an importable module. A product exposes a build result externally, such as a library or executable. Not every internal target must be a separate product.

Swift protocol. A declaration of requirements a conforming type must provide. It lets a caller use the contract without depending on one concrete implementation. Compiler-checked conformance does not establish every behavioural obligation.

Dependency injection. Receiving the collaborators a component needs from outside it. Whoever assembles the system provides a conforming implementation; the using component's responsibilities remain independently checkable.

Module and process. A module organises code and accessible names. A process is a running program that can terminate separately from other programs. A library loaded by an application does not thereby become an independent process.

Broker. An application responsibility admitting or denying operations under the path's checks. It need not be a separate process or another model.

Revision. The identity of a particular version. Its value depends on how it is assigned and connected to content.

Fixture. A curated input representing a reproducible case. It is distinct from outputs produced during testing.

Diff and commit. A diff shows differences between versions. A commit identifies a revision in Git history. Neither alone establishes runtime behaviour.

Idempotency. The property of an operation that, under the specified contract and identity, can be repeated without adding an unwanted equivalent effect. It must be verified on the component producing the effect.

Inference. Using a model's already learned parameters to process a request and produce a response. Retrieving a note changes available material without implying parameter updates.

Verifiable quotation. A reference allowing a source and passage to be found under stated conditions. Its existence does not alone establish support for the conclusion.

Asynchronous operation, `async` and `await`. An asynchronous operation can suspend and resume. `async` declares participation in Swift's mechanism; `await` marks a possible suspension point for the caller. They do not require a new thread or parallel execution.

Task and structured concurrency. A task represents asynchronous work without implying agentic capabilities. In structured concurrency, child activities such as those started with `async let` have a lifetime governed by their containing scope. A task created with `Task { ... }` instead requires explicit lifetime management in the path considered.

Actor and `MainActor`. Isolation domains coordinating state access under Swift rules. `MainActor` is shared and suitable for interface state on the main execution path. Isolation protects access; it does not itself choose which result remains relevant.

Reentrancy. The possibility of other admitted work on the same actor progressing while an operation is suspended. State may have changed after the wait, so an earlier condition may no longer hold.

Attempt generation. An identity recognising work still admitted to modify particular state. In the laboratory it is a UUID; it is neither a document revision, permission nor model version.

Cooperative cancellation. A request to interrupt a task that the work must react to. It guarantees neither immediate termination nor reversal of external effects already produced. Invalidating a generation permits rejection of late deliveries even if the service does not cooperate.

Sendable. A Swift protocol concerning transfer of values across concurrency boundaries under required checks. Conformance certifies neither data currency nor the access required to use them.

Provider, model and endpoint. The provider offers the service; the model identifies the requested capability; the endpoint is the destination receiving a

particular request type. API format and admitted capabilities must be checked together: changing a name does not establish compatibility.

SSE, Server-Sent Events. A format delimiting text events within a stream. In the studied subset, the reader reconstructs data: lines and recognises an event's end through a blank line. Received chunks need not coincide with complete events or characters.

Delta and complete response. A delta is a partial update. Provider completion is a separate event requiring valid content. Neither replaces assignment completion criteria.

`call_id`. The identifier of a particular tool call in the protocol. It connects proposal and result; it is neither chat identity, source identity nor a guarantee of durable idempotency.

Tool observation. The identified outcome of an operation: it may include acquired data, denial or a limit. Its authority depends on its producing component. It supplies the loop's continuation without automatically becoming an instruction to execute.

Partial deliverable. A result keeping unmet conditions and available material recognisable. In the laboratory, it requires missing sources consistent with read outcomes; adding “partial” to an invented conclusion is insufficient.

Contextualised reading

From Prompt to Software develops the AI-assisted working method: context, acceptance criteria, increments, reuse, tests and programmer responsibility. That book's cases are distinct from Velia's; its practices are applied here to the real project. The source link leads to its Italian project materials.

The Swift Programming Language is the language's primary reference. This collection introduces structures, protocols and visibility when problems need them; chapter 6 adds concurrency, isolation and state ownership. The **PackageDescription documentation** develops manifest construction, distinct from assistant behaviour.

Official Swift proposals **SE-0304**, **SE-0306** and **SE-0316** develop structured concurrency, actors and global actors respectively. They are language proposals explaining mechanisms used, rather than qualifications of Velia's behaviour. SE-0316, by John McCall and Doug Gregor, specifically clarifies the shared isolation model to which `MainActor` belongs.

Official OpenAI documentation on **streaming responses**, **function calling** and **production practices** accompanies chapters 7 and 8. The **WHATWG SSE standard**

distinguishes the general format from the service contract. These references describe mechanisms and rules; they are neither laboratory results nor product qualifications.

Research sources and reports about the OpenAI-Hugging Face episode are presented in the introduction with their scope and limits. They are not results from the Swift laboratory. The source guide distinguishes those references from code revisions and executable material.

Reading and reproducing the Velia collection

This collection includes the introduction and chapters 1-8 of *Building an Agentic AI, Brick by Brick*. It follows the experience of Velia in the Jarvis project and enters its Swift architecture. Document cases are invented for the book; the referenced components, package manifest and chapter 3's recovery change come from the real repository.

The reading path starts with the introduction, followed by the assignment, the three architectures, the recovery change, the first Swift laboratory, the component map, concurrency and state, transport and the first loop. Solutions and glossary accompany the collection. This order builds prerequisites; it does not reproduce a complete development chronology.

Which code the book describes

Velia's general reference is commit `7d0c7fe19a66982c50bb8e4deef51a3dfcb698b2` on Jarvis's dev branch. A commit identifies a revision in Git history. The book does not implicitly examine the customisation branch or the application installed on the computer. Architectural documentation dated 26 October 2026 describes an earlier snapshot and is used for orientation, compared with sources at the adopted reference.

For chapter 3's episode, we compare commit `a44709390ca79ca4a700dea78e2e485a739fa38a` with its parent. The parent is the immediately preceding revision in that commit's history. Project instructions and the diff connect the need for model-led interpretation to the code transformation; they provide neither a programming-dialogue transcript nor model-evaluation results.

Responsibility	Source in the Jarvis repository	Use in the collection
Request, context and coordination	Sources/VeliaDesktop/ChatAgentCoordinator.swift and AssistantController.swift in the same directory	Model/application boundary; candidate, relevance and recovery in chapter 3.
Model transport	ChatModelTransport and OpenAIChatModelTransport in Sources/VeliaDesktop/ChatAgentCoordinator.swift	Contract received by the coordinator, distinct from the core module.

Responsibility	Source in the Jarvis repository	Use in the collection
Tool execution and store	Sources/VeliaCore/AgentRuntime.swift, AgentStore.swift and AgentDatabase.swift in the same directory	Concrete runtime, state access and SQLite persistence in chapter 5.
Runs and context history	Sources/VeliaCore/ChatRuns.swift and AgentStore.swift in the same directory	Distinction between conversation and recorded work.
Notebook	Sources/VeliaCore/ChatNotebook.swift	Different sections and bases for notes, sources and hypotheses.
Inventory and coverage	Sources/VeliaDesktop/FolderInventory.swift	Distinction between found, acquired and returned, without attributing semantic understanding.
Modules and products	Package.swift	Seven targets, five products and dependencies represented in chapter 5.
Auxiliary processes	Sources/VeliaTerminalHost/main.c and Sources/VeliaPythonHost/main.c	C supervisors, distinct from modules loaded in the desktop and from the model.
Conversations and concurrent attempts	Sources/VeliaDesktop/ChatWorkspace.swift and AssistantController.swift	Per-chat controllers, selection, generation, updates and stopping in chapter 6.

The excerpt manifests for chapter 7 and chapter 8 identify sources, lines and fragment hashes. The chapter 6 excerpt manifest also identifies educational-source lines. Retained excerpts identify revision, source and reading point. These are fragments that may be reformatted in prose, rather than standalone compilable files. The complete Jarvis repository is unnecessary to run this collection's laboratories and is not implicitly distributed with them.

The previous introduction's source is retained as Italian editorial material. The new introduction inherits memories, episode, distinctions and developed comparison, updating connections and the path to Velia and Swift. The historical Centrale operativa reference retains its role as a comparative laboratory.

Which programs we can run now

The collection contains five complete programs, distinct from the Velia product. The first is the source verifier. It requires Swift 6 and Python 3, using neither network nor credentials. It compiles in private space, runs curated fixtures and removes compiled files and caches. It verifies limited reference properties; it neither runs a reduced Velia nor evaluates model understanding.

From the repository root:

```
python3 quality-gates/storage/gate.py run -- \
  python3 ai-agentica/esempi/velia/capitolo-04/esegui.py \
  --verifica
python3 quality-gates/storage/gate.py check
```

frase-non-sostenuta accepts the references while leaving evaluation of the conclusion open. Velia application verification remains distinct from this laboratory.

The second program is the component-boundary package. It also requires Swift Package Manager, included in the Swift tools, and has no external dependencies. Three targets separate contracts, core and application. In-memory and file adapters pass through the same identity and revision check; the program does not repeat chapter 4's quotation checking.

```
python3 quality-gates/storage/gate.py run -- \
  python3 ai-agentica/esempi/velia/capitolo-05/esegui.py \
  --verifica
python3 quality-gates/storage/gate.py check
```

The driver compiles once in a private copy and compares eight cases with prepared criteria. It does not compile Jarvis's package or establish compatibility of its macOS distribution. Earlier Italian collections 1-4, 1-5 and 1-6 remain reading snapshots; the current reference includes chapters 7 and 8 and updated supporting material.

The third program is the late-response experiment. It requires Swift 6 and Python 3, with no external packages. It uses one MainActor controller per chat, immutable chat-and-generation references and an actor delivering answers in prepared order. A separate counterexample shows that isolation alone does not prevent a logically superseded overwrite. Two reads with `async let` show a structured scope without measuring parallelism.

```
python3 quality-gates/storage/gate.py run -- \
  python3 ai-agentica/esempi/velia/capitolo-06/esegui.py \
  --verifica
python3 quality-gates/storage/gate.py check
```

The eight cases check two orders for chat switching and replacement, the current answer, stopping, the counterexample and independent reads. They do not synchronise through assumed delays. The service deliberately ignores cancellation to provoke late delivery; that condition is not a transport contract for the real application.

In the real source, `ChatWorkspace.selectLoaded` retains the controller of the chat being left; connections created by `ensureCollaborationController` capture chat identity. `AssistantController.append`, final and error paths, and `cancelResponse` let us follow generation checking. `AgentRuntime` and `AgentStore.finishStep` distinguish waiting, admission of completion and consumption recording, which may occur after stopping too. Manifest-identified excerpts retain these steps' context; they are not standalone files to compile.

Streamed requests and the chapter 7 and 8 loop

The fourth program is the stream reader: it constructs the essential request body and receives curated SSE bytes through prepared transport. Thirteen cases check fragmentation, identity, completion, refusals, errors and limits. It opens no HTTP connection, receives no credentials and does not establish account compatibility. The fifth is the loop, reusing that reader and returning identified observations to the next turn. Its nine sequences check reads, denied access, contract, duplicates and stopping at the limit; they include a semantically false conclusion passing reference checks alone.

```
python3 quality-gates/storage/gate.py run -- \
  python3 ai-agentica/esempi/velia/capitolo-07/esegui.py \
  --verifica
python3 quality-gates/storage/gate.py run -- \
  python3 ai-agentica/esempi/velia/capitolo-08/esegui.py \
  --verifica
python3 quality-gates/storage/gate.py check
```

The real reader is `ChatAgentStreamParser` in `Sources/VeliaDesktop/ChatAgentCoordinator.swift`.

`OpenAIChatModelTransport.complete` in the same file describes `URLSession`, headers, HTTP checks and cancellation; `OpenAIRedirectPolicy` in `OpenAIClient.swift` refuses redirects. `KeychainTokenStore`, in the same client file, and `SystemKeychainStorage` in `TouchIDCredentials.swift` identify credential responsibility. The book does not access stored values.

For the loop, read body construction, `chat_model` execution, complete output, call identity and appending `function_call_output` in

`ChatAgentCoordinator.respondWithMemory`. Transition to complete or partial is an application decision. `AgentRuntime.execute` resolves capability and schema before the step; store and handler apply their respective boundary checks. The educational loop has in-memory state and at most three turns: those limits are not transferred to the product.

The official sources consulted for the Italian edition on 28 October 2026 are OpenAI's guides to **streaming**, **function calling**, **conversation state** and **production practices**, and the **WHATWG SSE standard**. They describe contracts and format, distinct from the Git snapshot and program tests. Fixtures are curated inputs; no recording of tests or of a service is retained in the collection.