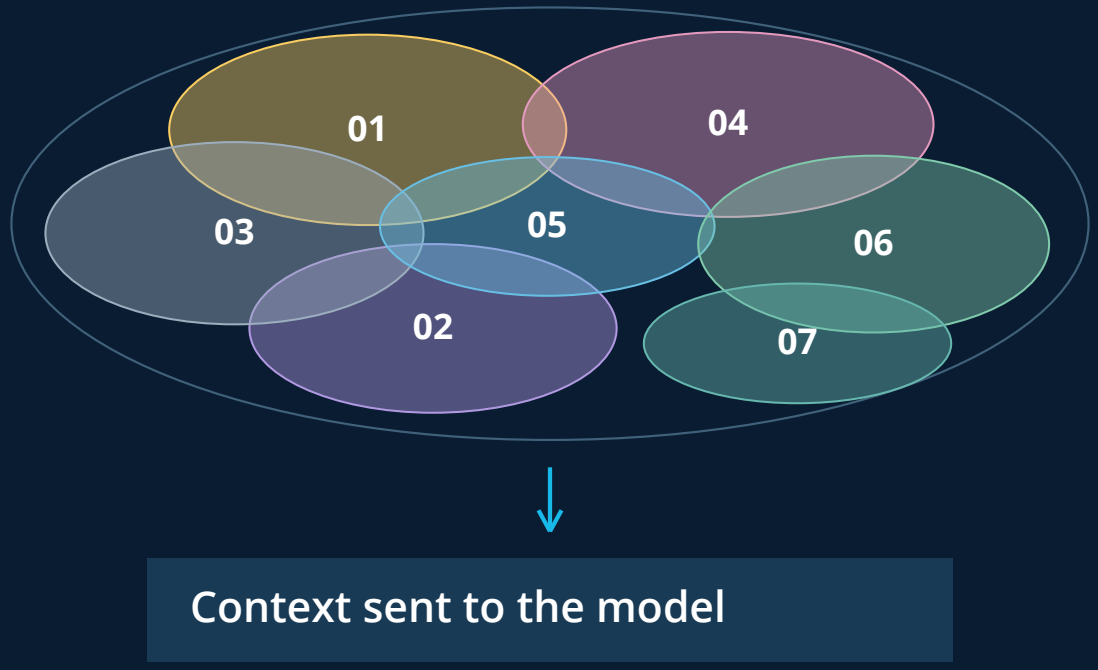
XAVKI

PART I / THE MODEL'S CONTEXT

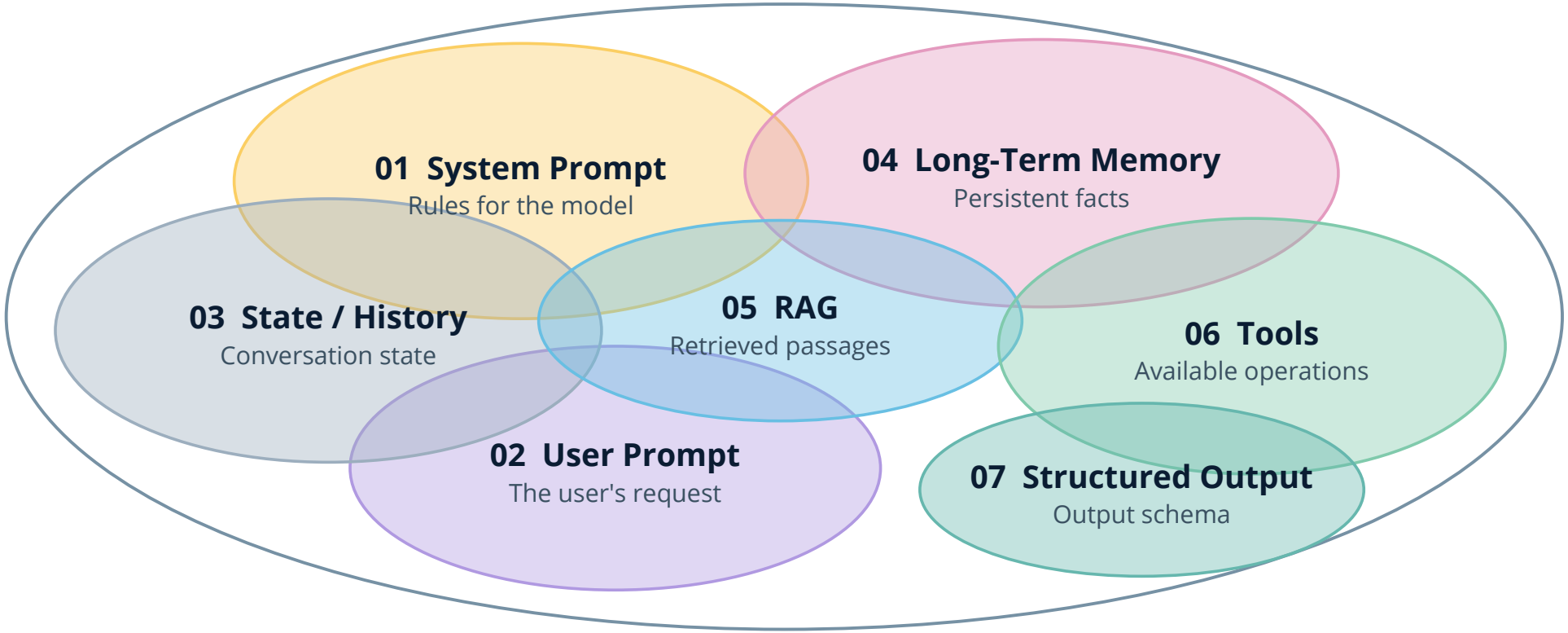
Context Engineering

- Select the information the model needs
- Set instructions for the current turn
- Describe the tools it can call



The seven components of context

The outer ellipse shows what the model receives for one call. Its contents depend on the task and the current turn.



Adapted from Philipp Schmid, “The New Skill in AI is Not Prompting, It's Context Engineering,” June 30, 2025.

www.philschmid.de/context-engineering

System Prompt - Set the model's ground rules

Use a system message for instructions that apply across requests.

```python
import os
from mistralai.client import Mistral

client = Mistral(api_key=os.environ["MISTRAL_API_KEY"])
model = "mistral-small-latest"

system = {
    "role": "system",
    "content": (
        "Distinguish documented values from observed "
        "values. Flag any unknown value."
    ),
}
```

Mistral Python SDK

SET UP THE CLIENT

- Install mistralai
- Set MISTRAL_API_KEY
- Reuse the client and model

PLACE THE INSTRUCTIONS

- Put shared rules in system
- Send it first in each call

User Prompt - Send the request to the model

Put the current request in a user message.

```
question = (
    "Compare the documented and active timeouts "
    "for the api service."
)
user = {"role": "user", "content": question}
messages = [system, user]

reply = client.chat.complete(
    model=model,
    messages=messages,
)
answer = reply.choices[0].message
```

Mistral Python SDK

INPUT

- Message order: system, then user
- The role field identifies the sender

OUTPUT

- Reply in the assistant message
- Both timeout values are still unknown

State / History - Assemble the next turn

The application selects which messages to send with each call.

```
follow_up = {
    "role": "user",
    "content": "Start with the documented value.",
}

next_messages = [*messages, answer, follow_up]
second = client.chat.complete(
    model=model,
    messages=next_messages,
)
```

Mistral Python SDK

ROLE SEQUENCE

- system → user → assistant → user
- The last message clarifies the request

IN THE FOLLOWING SNIPPETS

- Start with system and user
- For tool use, also keep the assistant's call and the tool result

Long-Term Memory - Recall a user preference

Retrieve a saved preference when preparing a model call.

```
row = conn.execute(
    "SELECT locale FROM preferences "
    "WHERE user_id = %s",
    (user_id,),
).fetchone()

context = [system]
if row:
    context.append({
        "role": "user",
        "content": f"Preferred language: {row[0]}",
    })
context.append(user)
reply = client.chat.complete(model=model, messages=context)
```

Mistral Python SDK

APPLICATION

- Connect to PostgreSQL with psycopg
- Get user_id from the authenticated session
- Store a validated language code in locale

MODEL INPUT

- Add the preference if found
- Otherwise, send only system and user

XAVKI

RAG - Find relevant passages with pgvector

Use the same embedding model for the question and the passages.

```
import json

embedding = client.embeddings.create(
    model="mistral-embed", inputs=[question],
).data[0].embedding

sql = '''
SELECT id, content
FROM passages
WHERE corpus_id = %s
ORDER BY embedding <=> %s::vector
LIMIT 3
'''
rows = conn.execute(
    sql, (corpus_id, json.dumps(embedding)),
).fetchall()
```

PREREQUISITES

- PostgreSQL with pgvector
- Vectors with matching dimensions
- Filter by the authorized corpus

RETRIEVAL

- <=> computes cosine distance
- Sort by increasing distance
- Return up to three passages

pgvector documentation

RAG - Add retrieved passages to messages

Add the retrieved text and passage IDs to the model's input.

```
passages = [
    {"id": str(doc_id), "text": text}
    for doc_id, text in rows
]
payload = json.dumps(
    {"question": question, "passages": passages},
    ensure_ascii=False,
)
rag_messages = [
    system,
    {"role": "user", "content": payload},
]
reply = client.chat.complete(
    model=model, messages=rag_messages,
)
```

pgvector documentation

WHAT THE MODEL RECEIVES

- The question and each passage's text and ID
- The vector is used during retrieval

WHAT THE EVIDENCE SHOWS

- Service guide: documented timeout
- Live configuration: active timeout

Tools - Define a tool in the SDK

The tools field describes the function and the arguments it accepts.

```
tool = {
    "type": "function",
    "function": {
        "name": "read_timeout",
        "description": "Reads the active timeout in seconds.",
        "parameters": {
            "type": "object",
            "properties": {
                "service": {"type": "string", "enum": ["api"]}
            },
            "required": ["service"],
        },
    },
}
```

Mistral docs · Function calling

TOOL CONTRACT

- Function: read_timeout
- Required argument: service
- Allowed value in this example: api

APPLICATION

- Implement the configuration lookup
- Check the arguments before execution

Tools - Inspect the model's tool call

A tool call identifies the function, serialized arguments and call ID.

```
reply = client.chat.complete(
    model=model, messages=rag_messages, tools=[tool],
    tool_choice="any", parallel_tool_calls=False,
)
assistant = reply.choices[0].message
call = assistant.tool_calls[0]

# Example fields from the response
# call.id                -> "aB3kL9mQ2"
# call.type              -> "function"
# call.function.name     -> "read_timeout"
# call.function.arguments -> '{"service":"api"}'
```

NUMBER OF CALLS

- any requires a tool call
- parallel_tool_calls=False allows one call at a time

CALL ID

- Keep call.id
- Use it to match the result to this call

Mistral docs · Function calling

XAVKI

Tools - Return the result with the tool_call_id

Send the assistant's tool call first, followed by the tool's result.

```
args = json.loads(call.function.arguments)
valid_args = args == {"service": "api"}
if call.function.name != "read_timeout" or not valid_args:
    raise ValueError("Unexpected tool call")

seconds = active_config[args["service"]]["timeout_s"]
result = {
    "role": "tool",
    "name": call.function.name,
    "tool_call_id": call.id,
    "content": json.dumps({"timeout_s": seconds}),
}
final = client.chat.complete(
    model=model, tools=[tool], tool_choice="none",
    messages=[*rag_messages, assistant, result],
)
```

LOCAL EXECUTION

- Validate the name and arguments
- Read timeout_s from active_config

NEXT TURN

- Include the guide passages and the observed value
- tool_choice="none" requests an answer without another tool call

Mistral docs · Function calling

Structured Output - Get a typed response

Define the response structure with a Pydantic class.

```python
from pydantic import BaseModel, Field

class Timeouts(BaseModel):
    documented_s: int | None = Field(ge=0)
    active_s: int | None = Field(ge=0)

facts = {"passages": passages, "active_s": seconds}
request = {"question": question, "facts": facts}
parsed = client.chat.parse(
    model=model,
    messages=[system, {"role": "user",
                       "content": json.dumps(request)}],
    response_format=Timeouts,
)
timeouts = parsed.choices[0].message.parsed
```

Mistral docs · Structured outputs

OUTPUT CONTRACT

- Both fields are required
- Integer ≥ 0, or null if unknown

VALIDATION

- Pydantic checks types and allowed values
- Verify the values against the guide and the observation

XAVKI

PART II / FROM CONTEXT TO EXECUTION

Harness Engineering

The software that runs the agent

- Prepare each model call
- Execute actions and keep their results
- Decide whether to continue or stop

PART I / WHAT THE MODEL RECEIVES

Messages, tools and an output schema

PART II / WHAT DRIVES EACH TURN

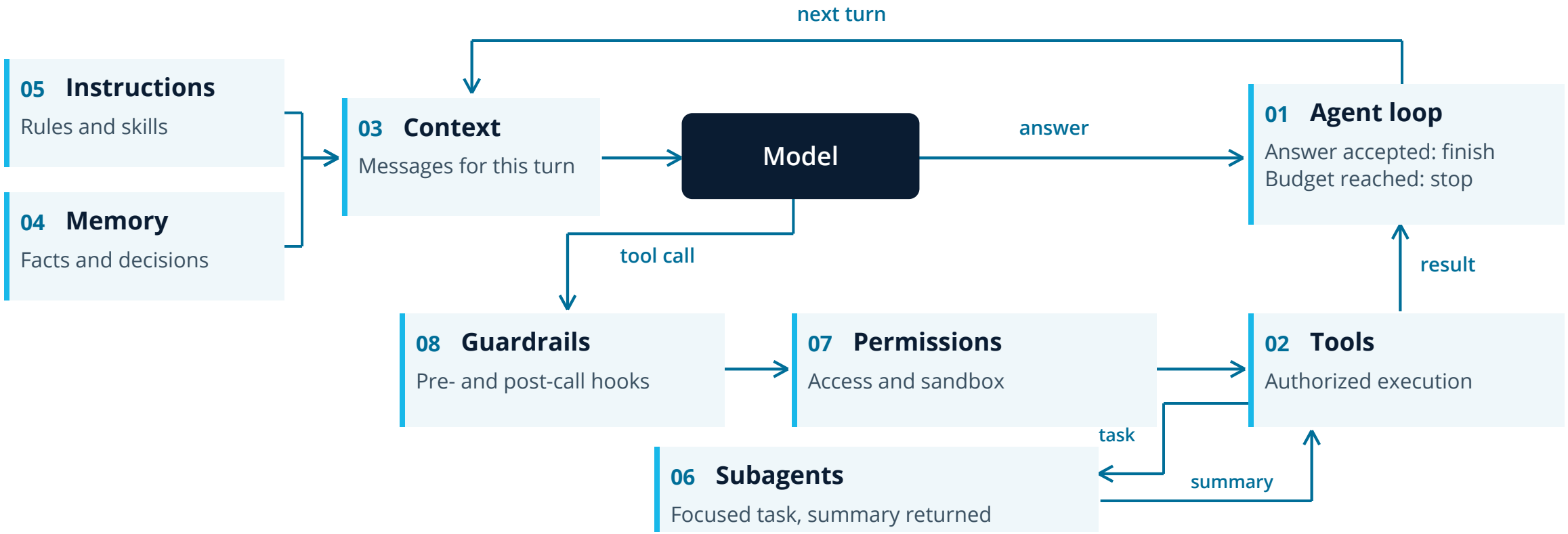
Context selection, execution and validation

Harness - Model and harness interdependence



Harness - The nine layers of an agent harness

Select a numbered layer to jump to its explanation.



Adapted from Shivani Virdi (Neosage), “The 9 layers of an agent harness.”

Skills - Load a procedure when the task calls for it

A skill packages reusable instructions for a particular task.

On every call

- Include the standing system instructions
- List available skills by name and purpose

On demand

- Select the skill that fits the task
- Add its procedure to the current context
- Expose only the tools authorized for that task

IN THE TIMEOUT EXAMPLE

Standing rule: flag unknown values.
Comparison procedure: read the guide, check the live configuration and compare the evidence.

PART I 03 System Prompt 08 RAG · messages 09 Tools · description

Context Assembly - Build the first model call

One question, two sources: the service guide and the live configuration.

Rebuild the request

- Start with system and user
- Identify the service from history
- Recall the profile language if relevant

Retrieve the guide

- Search only the authorized corpus
- Keep the passage text, ID and version

Expose the lookup tool

- Supply rag_messages
- Declare read_timeout in tools
- Wait for the result

If the guide gives no timeout, the documented value stays unknown, even when the live configuration supplies a value.

PART I 03 System Prompt 04 User Prompt 06 Long-Term Memory 07 RAG · retrieval 08 RAG · messages

Control Loop - Decide what happens after each reply

On each turn, execute a tool call, validate an answer or stop the loop.

```python
def run(messages, ask, execute, accept, max_turns=4):
    for _ in range(max_turns):
        reply = ask(messages)
        if reply.tool_calls:
            messages.append(reply)
            for call in reply.tool_calls:
                messages.append(execute(call))
        elif accept(reply):
            return "done", reply
        else:
            return "invalid_result", reply
    return "budget_exhausted", None
```

ASK → MODEL

- Call the model through the SDK
- When streaming, assemble the arguments before execution

EXECUTE → OBSERVATION

- Return one tool message per call
- Check the final answer against the evidence

> **Set separate limits for model turns, tool actions and elapsed time.**

PART I 04 User Prompt 10 Tools · tool_call 11 Tools · result 12 Structured Output

Tools - Turn a tool call into a verified observation

From call.function.arguments to an authorized configuration lookup.

1. Resolve and authorize	• Look up read_timeout · Decode arguments · Check access to the service
2. Execute	• Read the live configuration • For a code tool, isolate execution and limit resources
3. Attach the result	• Append assistant, then tool • Keep call.id, status and observation time
4. Handle errors	• Return timeout to the model • Leave the active value unknown

PART I 03 System Prompt 09 Tools · definition 10 Tools · tool_call 11 Tools · result

Hooks and Sandboxes - Control a tool call

A hook is a function the harness calls at a specific point during execution.

Before the call · hook	• Check the name and arguments • Block out-of-scope requests
Decision · permissions	• Allow, deny or request approval • Check the caller's identity and the target resource
During execution · sandbox	• For code tools, isolate files and limit network access and runtime • Grant only the access the task needs
After the call · hook	• Redact secrets before logging or returning results • Return the status with the tool_call_id

> **If the lookup is denied, return a denied status. Without a reading, the active timeout remains unknown.**

PART I 09 Tools · definition 10 Tools · call 11 Tools · result

Memory - Decide what to keep across turns and restarts

State can last for a turn, persist in a user profile or survive a run interruption.

Session

- Keep the current question and recent observations
- When summarizing, keep the service, guide version and unresolved question

Profile

- Reload the authenticated user's language
- Scope the preference to that user

Checkpoint

- Save the run_id, completed step and evidence references
- On resume, check whether the reading is still current

> **Before retrying a write, check whether it already took effect or reuse an idempotency key the service recognizes.**

Anthropic · Effective harnesses for long-running agents

PART I 05 State / History 06 Long-Term Memory 11 Tools · result

Workflows - Decide who controls the next step

Coordinate two lookups: the service guide and the live configuration.

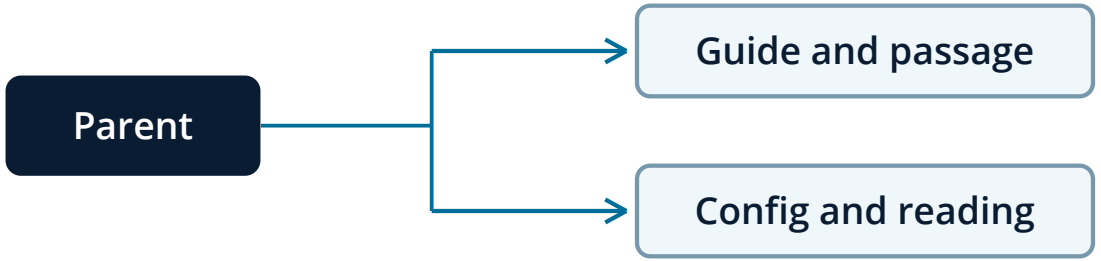
Explicit workflow

- Set the order of steps
- Define transitions and error handling



Delegation

- Call each subagent as a tool
- Assign a focused lookup
- Get a summary with supporting evidence



Before validation, match both pieces of evidence to the same service ID.

PART I 07 RAG · retrieval 11 Tools · result 12 Structured Output

Structured Output - Check the values after the schema

Two checks for Timeouts: does the response match the schema, and do its values match the evidence?

```python
def validate_active(output, observation):
    if observation["status"] != "ok":
        return output.active_s is None
    return output.active_s == observation["timeout_s"]
```

DOCUMENTED TIMEOUT

- Check documented_s against the correct guide version
- Keep the passage ID

AFTER A LOOKUP TIMES OUT

- active_s = 30 is a valid integer
- No reading: block publication

AFTER A SUCCESSFUL LOOKUP

- Observed value: 30 seconds
- Reject every other value, including null

> **Before publication, enforce the system instruction to flag unknown values.**

PART I 03 System Prompt 08 RAG · messages 11 Tools · result 12 Structured Output

Tracing - Explain an incorrect answer

Use one run_id to link the model's input, the tool call and the validation decision.

Turn 1 · preparation	• Requested service: api • Retrieved passage: guide:v7
Turn 1 · decision	• Call: read_timeout(service="api") • Link to the connector via tool_call_id
Lookup · observation	• Status: timeout • Original call ID retained
Turn 2 · validation	• Proposed value: active_s = 30 · No reading · Publication blocked

> **Diagnose missing evidence, failed lookups or validation errors. Measure time and cost per step.**

PART I 08 RAG · messages 10 Tools · tool_call 11 Tools · result 12 Structured Output

Evals - Replay a trace in CI

Freeze the test inputs: messages, guide passage and simulated observation.

Case A · successful lookup	• Reading: 30 s · Accept 30 · Reject any other value
Case B · lookup timed out	• Status: timeout · Accept null · Reject every integer
Compare two versions	• Replay the same inputs • Compare verdicts, unsupported values, tool calls and latency

> **In CI, block release when a required case fails. Then compare costs across a broader test set.**

OpenAI · Harness engineering and feedback loops

PART I 11 Tools · result 12 Structured Output

Ecosystem - Build, orchestrate, route and test

Four technical categories from Pinggy's overview.

Agent frameworks	• LangGraph · Model state transitions · Define stopping conditions
Workflow orchestration	• Prefect, n8n • Chain retrieval, the model call and validation
Hosting and routing	• OpenRouter · Route model requests · Manage state and actions in the application
Evaluation	• Promptfoo, DeepEval · Replay cases · Detect regressions

Pinggy · Eight families of harnesses

LangGraph · Overview

Promptfoo · Introduction

PART I 05 State / History 09 Tools 12 Structured Output

Ecosystem - Match the harness to the task

Four use cases from the same overview. A single product can support several.

Software development	• Codex CLI • Carry the diff, repository state and test results into the next turn
Content creation	• Video editing · Gather assets and the brief · Track revisions through review
Productivity	• Calendar and documents • Limit connector access and permitted actions
Entertainment	• Interactive storytelling • Retain player choices and character constraints

Categories adapted from Pinggy · AI Harness Engineering

PART I 03 System Prompt · 05 State / History · 06 Long-Term Memory · 09 Tools

XAVKI

A Complete Run - From a user message to a verified value

Original request: “Compare the documented and active timeouts for the api service.”

01 Prepare	• Load instructions and request · Retrieve the authorized guide · Keep passage IDs
02 Observe	• Receive the tool call · Authorize the lookup · Match the result to the call
03 Decide	• Compare the evidence • If the lookup times out, keep active_s = null
04 Verify and save	• Validate Timeouts and the evidence · Save state · Replay failures

> **When debugging a run: what evidence did the model receive, and what did the harness do next?**

PART I 04 Request 08 Passages 11 Observation 12 Typed response