PRODUCT DOCUMENT · AGENT FRAMEWORK FLOWS

Agent framework flows: signed, confined and recorded

How Saphan Studio runs a customer’s LangGraph flow as a governed order: a person signs it, the engine runs it confined on a seat in your installation, and the result is checked and recorded. The engine never accepts a result by itself.

PUBLIC EDITION

2	1	0	72 h
directions: Studio calls your flow, or your code starts one	person’s signature behind every flow run	results the engine accepts by itself	default wait; then an unanswered result expires

APPLIES TO

Team and Enterprise

LangGraph; other frameworks on the same flow runner

PUBLISHED

13 November 2026

Saphan Studio · saphan.ai/product

CONTENTS

01 Summary

02 Two directions

03 The flow-run order

04 The flow manifest

05 How a run is contained

06 Results, checks and acceptance

07 What your code uses

08 Examples, in Python

09 Scope

10 Before your first flow

11 What this document does not claim

12 The Saphan ecosystem and this document set

HOW TO READ THIS DOCUMENT

Section 1 is the summary. Sections 2 to 4 describe how a flow run starts: the two directions, the order a person signs and the manifest it names. Sections 5 and 6 cover what the boundary guarantees and how a result is checked and accepted. Section 7 is what your code uses and section 8 shows it in Python, end to end. Section 9 sets out the scope, and section 10 the questions that shape your first flows. The last section maps the Saphan ecosystem and the other documents.

SECTION 01

Summary

Saphan Studio runs agent-framework flows, LangGraph first, under the same governance as its coding agents. A flow run is an order: a person signs it, the engine places it on a governed seat in your installation, the flow runs confined, and its result is checked and recorded. Nothing is merged. The result is data.

The design keeps four things closed: a person's signature before anything runs, confinement while it runs, a record of every act, and no automatic acceptance. The details around them are yours to set: which direction starts a run, who accepts the result, what the manifest declares.

The engine never decides whether a result is good enough. A person accepts it, or a connector that the signer named sends a signed decision. The engine records which, when, and on what result.

Section 8 shows the calls in Python, end to end. Section 10 lists the questions that shape your first flows.

SECTION 02

Two directions

A flow can be started from Saphan Studio or from your own application. In both directions it runs inside Saphan, on a seat in your installation, and a person signs it first.

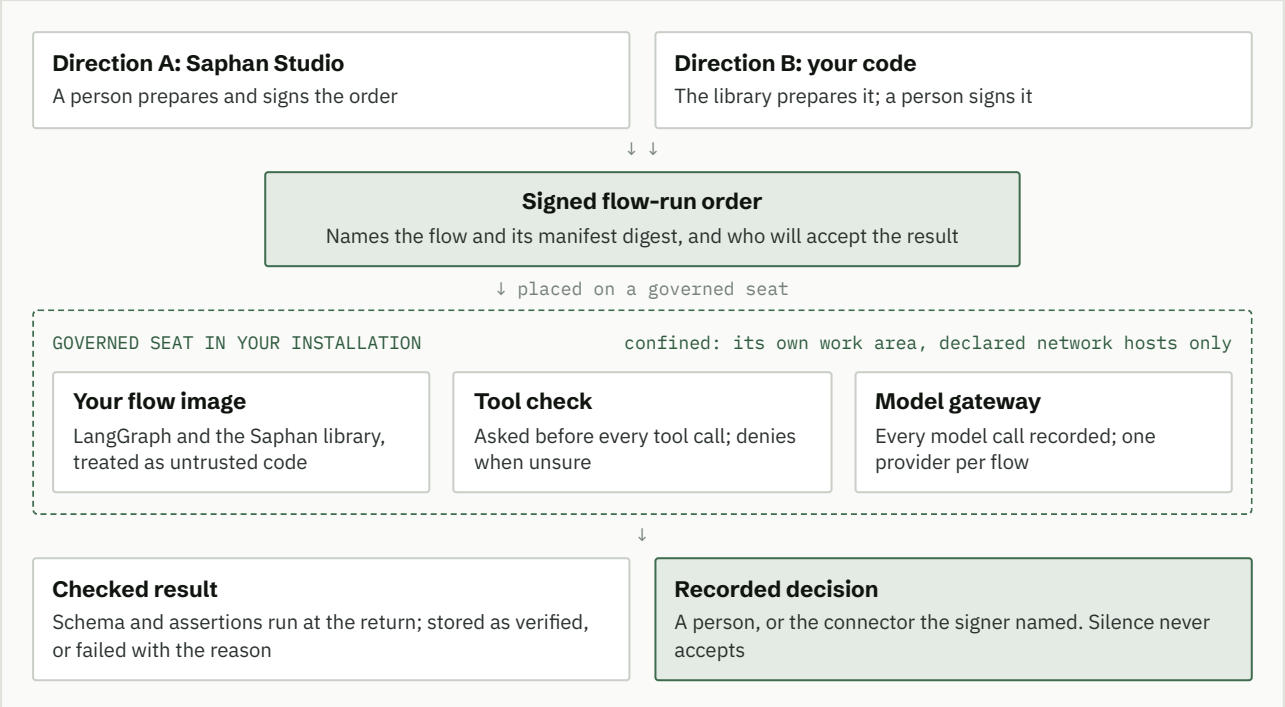


FIGURE 1 · ONE PATH FOR BOTH DIRECTIONS: SIGNATURE, CONFINED RUN, CHECKED RESULT, RECORDED DECISION

	DIRECTION A: STUDIO CALLS YOUR FLOW	DIRECTION B: YOUR CODE STARTS AN ORDER
Who prepares the order	a person in Saphan Studio	your code, through the Saphan library
Who signs	a person enrolled in your installation	the same: a person; your code cannot sign
Where it runs	a governed seat in your installation	a governed seat in your installation

	DIRECTION A: STUDIO CALLS YOUR FLOW	DIRECTION B: YOUR CODE STARTS AN ORDER
Who reads the result	people in the Studio review screen	your code reads the stored result
Typical use	a team runs a vetted flow as part of its work	an application asks for governed agent work

Your application never gets a way around the signature. It can prepare an order and read a result; it cannot run a flow by itself.

SECTION 03

The flow-run order

A flow run starts only from an order that a person signs. The order names one flow and says who will accept its result. It creates no branch and nothing to merge.

WHAT THE ORDER CARRIES	WHY
The kind: flow run	the engine applies the flow-run rules, not the rules for code changes
The flow's manifest and its digest	the signature then covers this exact manifest (section 4)
The inputs	declared in the manifest and checked before the run
Acceptance: `person` or `delegated: <connector>`	the signer decides who accepts, before anything runs
Acceptance wait (default 72 hours)	an unanswered result expires; it is never accepted by default

01 Your code, or a bot, may prepare and register an order. Only a person enrolled in your installation can sign it.

02 The signature covers every byte of the order, including the manifest digest and the acceptance line.

03 A flow-run order runs only on the governed flow runner. A request to run it on an unconfined runner is refused by name.

04 Delegated acceptance exists only for flow-run orders. On any other kind of order it is refused when it is signed and again when it is placed.

SECTION 04

The flow manifest

The manifest is the flow's declaration: what runs, what it may reach, and what it must return. It sits in your installation's shared record, next to the orders that use it.

FIELD	WHAT IT DECLARES
Image	the container image, pinned by digest; a tag alone is refused
Framework	LangGraph and its version

FIELD	WHAT IT DECLARES
Model provider	one per flow; every model call goes through Saphan's model gateway
Network	the hosts the flow may reach; everything else stays closed
Tracing	off by default; Saphan's recorded session is the trace
Inputs	the contract the order's inputs must meet
Output	the result's schema and the assertions checked at the return
Secrets	names only; the values stay with the seat and never enter an order or a manifest
Decisions	the bounded choices a person may answer when a flow pauses (section 9)

How the manifest is bound to the order

The order carries the manifest's digest. The engine reads the manifest as it was recorded when the order was registered, never a later copy, and compares the digest when the order is placed and again when the run starts. A mismatch or a missing manifest is refused by name, and so is a manifest that fails its own checks: an unknown field, an image without a digest, a second model provider, an undeclared tracing host.

A changed manifest needs a new signed order. The signer reviews the manifest with the order, or registers it once and reuses it by digest.

SECTION 05

How a run is contained

Your flow image is treated as untrusted code, however well you know it. These are the guarantees of the boundary, not its construction.

GUARANTEE	WHAT IT MEANS FOR YOU
Confined run	the flow writes only in its own work area; it touches no repository and no other seat
Declared network only	outbound traffic is closed by default; only the manifest's hosts are reachable
Recorded model calls	every model call passes the model gateway and is kept with the session
Secrets by name	values reach the flow at run time from the seat; configured values are scrubbed from its events
Tool check	before each tool call the library asks Saphan; a refusal, a timeout, a missing or unreadable answer all mean deny
Engine owns the lifecycle	the flow reports progress in Saphan's event format but cannot declare its own start, end or outcome
Bounded output	event lines, the event queue and tool-check input have size limits; past a limit the engine refuses by name

The tool check helps a well-behaved flow stay inside its grant. It is not the boundary: a flow can call a tool without asking. Confinement, the closed network and the model gateway are the boundary.

Progress events stream to the Saphan console as the flow runs. The run's outcome comes from the flow's exit and the result checks, never from what the flow says about itself.

SECTION 06

Results, checks and acceptance

Saphan checks every result and records who accepted it. The engine itself never accepts.

01 **Return.** The flow writes its result and artifacts. The engine checks the result against the manifest's schema and assertions and stores verified, or failed with the field or assertion.

02 **Default: a person accepts.** The result waits in the review screen with the check verdict beside it. A person may accept a failed result; the failure stays visible.

03 **Optional: a connector accepts.** If the signer chose delegated acceptance, your connector reads the stored result and sends a signed decision: accept or reject, with a reason, bound to that run and that exact result. Your code decides how: a person in your own console, or an automatic rule.

04 **Silence never accepts.** When the wait ends with no decision, the result is marked expired and stays not accepted. A person can still accept it.

05 **The record shows both facts.** The check verdict and the decision stand side by side. A connector's note on who decided is shown as the connector's claim, never as a verified identity.

The connector's identity

A connector is enrolled in your installation by your administrator, with its own key, an expiry, and revocation. Its key can sign one thing only: acceptance decisions for flow-run orders that name it. The review screen shows only what the engine stored at the return; nothing is recomputed when someone opens it.

WHAT THE ENGINE REFUSES	REFUSAL
A decision signed by another key, or for another run or another result	`acceptance-decision-unbound`
A second decision for the same run	`acceptance-decision-duplicate`
Delegated acceptance on any order that is not a flow run	`acceptance-delegation-not-allowed`
A connector that is not enrolled	`acceptance-connector-unknown`
No decision before the wait ends	`acceptance-expired` (not accepted)

Every refusal carries its reason and the next action, as everywhere in Saphan Studio.

SECTION 07

What your code uses

A Python library connects a flow to Saphan. It has no hidden defaults: your code passes the path of the Saphan command-line tool, the workspace, the transport and the time limits from its own configuration.

PIECE	WHAT IT DOES
Event emitter	writes Saphan’s progress events, one JSON line each; refuses undeclared fields and any claim about start, end or outcome; scrubs configured secret values
Tool-check client	asks Saphan before a tool runs, from the flow’s own work area; allows only on a clean answer; fails closed
Order client	prepares orders and reads the signing queue and replies as JSON; refuses by name what a remote client may not do
LangGraph middleware	wires the three into a graph
Acceptance connector client	reads the stored result and sends a signed decision; ships with two examples, a console for a person and an automatic rule, and no default policy

Limits are settings with defaults: 64 KiB per event line, 64 KiB per tool-check answer, 1 MiB of command output. A child process that exceeds a limit or its deadline is stopped and the call fails closed.

Your graph must treat a returned result as data, never as instructions to its model.

SECTION 08

Examples, in Python

One flow, end to end: an accounts-payable team at an example company, Acme, triages open invoices with a LangGraph flow.

EXAMPLE 1

The manifest

Stored in your installation's shared record at `flows/invoice-triage/MANIFEST.json`. Every field is checked before a run; a tag instead of a digest, a second model provider or a tracing host is refused.

```
{
  "schema_version": 1,
  "name": "invoice-triage",
  "framework": "langgraph",
  "framework_version": "1.0",
  "runtime": {
    "kind": "oci-image",
    "image": "registry.acme.example/flows/invoice-triage@sha256:58875b64d49f2609e46c77026073efae925d358f8f
afbdc6a0285aa9fa2923d3",
    "entrypoint": ["python", "-m", "invoice_triage"]
  },
  "models": [{"provider": "anthropic", "model": "claude-sonnet-4-5", "codec": "claude-messages"}],
  "egress": {
    "hosts": [{"host": "erp.acme.example", "port": 443, "why": "read open invoices"}],
    "workloads": []
  },
  "secrets": ["ERP_API_TOKEN"],
  "inputs": {
    "schema": {"type": "object", "required": ["batch_date"],
               "properties": {"batch_date": {"type": "string", "format": "date"}}}
  },
  "outputs": {
    "schema": {"type": "object", "required": ["invoices", "summary"]},
    "assertions": ["len(/invoices) <= 500", "/summary/flagged <= /summary/seen"]
  },
  "decisions": [],
  "telemetry": "off"
}
```

EXAMPLE 2

The order a person signs

The `flow:` line is the SHA-256 of the manifest file. Change one character of the manifest and the run is refused with `flow-manifest-digest-mismatch`.

```
# ORDER — run-invoice-triage-2026-11-12
- kind: flow-run
- flow-manifest: flows/invoice-triage/MANIFEST.json
- flow: sha256:ce18aca83b84540a8684187984bc8af69f550a18ba34b7a2d2e736cc631e709f
- inputs: .saphan/work/run-invoice-triage-2026-11-12/inputs.json
- acceptance: delegated:acme-ap-console
- acceptance-wait: 48h
- cap-usd: 6.00
- wall: 30m
```

EXAMPLE 3

Progress events from inside the flow

The emitter writes Saphan's event lines and refuses anything a flow may not claim.

saphan_agent.events

```
import os
from saphan_agent.events import Emitter

events = Emitter(
    scrub_values=[os.environ["ERP_API_TOKEN"]],   # exact values to redact from every event
    child_mode=True,                              # a flow may not claim its start, end or outcome
)

events.emit("loop.iteration", iteration=1, detail="fetching open invoices for 2026-11-12")
events.emit("loop.tool", tool="erp.list_invoices", detail="status=open, 214 found")
events.emit("loop.refusal", refusal_class="invoice-unreadable",
            detail="INV-20391: the PDF has no text layer")
```

WHAT THE CONSOLE RECEIVES

```
{"event":"loop.iteration","iteration":1,"detail":"fetching open invoices for 2026-11-12"}
{"event":"loop.tool","tool":"erp.list_invoices","detail":"status=open, 214 found"}
{"event":"loop.refusal","detail":"INV-20391: the PDF has no text layer","class":"invoice-unreadable"}
```

WHAT THE EMITTER REFUSES

```
events.emit("loop.finished", outcome="done")
# ValueError: loop-event-reserved: child cannot set lifecycle or outcome.
#             NEXT — let the adapter close the run

events.emit("loop.tool", tool="erp.list_invoices", cost_usd=0.02)
# ValueError: loop-event-field-unknown: event carries undeclared fields.
#             NEXT — use loop.Event fields
```

EXAMPLE 4

Asking before a tool runs

`RUNNER` stands for the values the flow runner hands to the flow at start: the hook binary, the run's grant, the work area and the input and result paths. Your code never chooses them.

saphan_agent.hook

```
from saphan_agent.hook import HookClient

gate = HookClient(
    binary=RUNNER.hook_binary,      # absolute path, supplied by the flow runner
    role=RUNNER.role,               # the run's grant, supplied by the flow runner
    mode=RUNNER.mode,
    timeout_seconds=2.0,            # no answer in time means deny
)

verdict = gate.tool_before("Write", {"file_path": "/etc/hosts", "content": "..."},
                           cwd=RUNNER.workdir)
verdict.allowed          # False: the path is outside the run's work area
verdict.refusal_class    # "tool-before-denied"
verdict.reason           # the engine's reason, with its next action

verdict = gate.tool_before("Write", {"file_path": f"{RUNNER.work_area}/INV-20391.json",
                                     "content": "{...}"}, cwd=RUNNER.workdir)
verdict.allowed          # True: inside the run's work area

# No hook binary, a crash, a timeout or an unreadable answer: always deny.
# Verdict(allowed=False, refusal_class="tool-before-unavailable", ...)
```

EXAMPLE 5

Both, inside a LangGraph graph

The LangGraph middleware does this wrapping for every tool. Shown by hand, so you can see each call.

saphan_agent.events + saphan_agent.hook + langgraph

```
import json
from typing import TypedDict
from langgraph.graph import StateGraph, START, END

class Triage(TypedDict, total=False):
    batch_date: str
    invoices: list
    summary: dict

def checked(tool, fn):                      # every tool call asks Saphan first
    def call(**args):
        verdict = gate.tool_before(tool, args, cwd=RUNNER.workdir)
        if not verdict.allowed:
            events.emit("loop.refusal", refusal_class=verdict.refusal_class,
                        tool=tool, detail=verdict.reason)
            raise PermissionError(verdict.reason)
        events.emit("loop.tool", tool=tool)
        return fn(**args)
    return call

list_invoices = checked("erp.list_invoices", erp.list_invoices)   # erp: your own client

def triage(state: Triage) -> Triage:
    open_invoices = list_invoices(status="open", date=state["batch_date"])
    flagged = [i for i in open_invoices if needs_review(i)]          # your model calls go here
    return {"invoices": flagged,
            "summary": {"seen": len(open_invoices), "flagged": len(flagged)}}

graph = StateGraph(Triage)
graph.add_node("triage", triage)
graph.add_edge(START, "triage")
graph.add_edge("triage", END)

with open(RUNNER.inputs) as f:
    result = graph.compile().invoke(json.load(f))
with open(RUNNER.result, "w") as f:                # RESULT.json in the work area
    json.dump(result, f)
```

EXAMPLE 6

Direction B: reading the queue and replies from your application

Your application reads what waits for a signature and the replies to an order, as JSON.

saphan_agent.cli

```
from saphan_agent.cli import SaphanCLI

saphan = SaphanCLI(
    binary="/usr/local/bin/saphan",     # your installed Saphan command-line tool
    workspace="/srv/saphan/acme",       # your workspace
    transport="local",                  # "local" on the Saphan host, "thin" through the Saphan gateway
    timeout_seconds=30,
)

queue = saphan.pending()                # orders waiting for a person's signature
print(len(queue["pending"]), "orders wait for a signature")

for reply in saphan.replies("run-invoice-triage-2026-11-12"):
    print(reply)                        # replies to that order, as JSON
```

EXAMPLE 7

Direction B: preparing a flow-run order

Your code registers the order; a person signs it in Saphan Studio. There is deliberately no `sign()` for your code to call.

saphan_agent.cli

```
order = saphan.flow_order(
    manifest="flows/invoice-triage/MANIFEST.json",
    inputs={"batch_date": "2026-11-12"},
    acceptance="delegated:acme-ap-console",   # or "person", the default
    acceptance_wait="48h",
    cap_usd=6.00,
)
order["path"]    # the order file, registered and waiting for a person's signature
order["flow"]    # the manifest digest the signature will cover
```

EXAMPLE 8

An acceptance connector

An automatic rule: accept a verified result, reject anything else. A console for a person makes the same `decide()` call with `decided_by={"kind": "person", ...}`. The engine stores the decision and shows `decided_by` as the connector's claim.

saphan_agent.acceptance

```
from saphan_agent.acceptance import AcceptanceClient

acceptor = AcceptanceClient(
    binary="/usr/local/bin/saphan",
    workspace="/srv/saphan/acme",
    key="/etc/acme/saphan/acme-ap-console.key",   # the connector's enrolled key
    timeout_seconds=30,
)

result = acceptor.result("run-7f3a2c")            # the projection stored at the return
# {"run": "run-7f3a2c", "order": "run-invoice-triage-2026-11-12",
#  "digest": "sha256:...", "check": "verified", "artifacts": ["RESULT.json", "RESULT.md"],
#  "acceptance": {"mode": "delegated:acme-ap-console", "deadline": "2026-11-14T09:00:00Z"}}

if result["check"] == "verified":
    acceptor.decide(result["run"], result["digest"], "accept",
                    reason="schema and assertions verified",
                    decided_by={"kind": "automaton", "label": "ap-auto-rule v3"})
else:
    acceptor.decide(result["run"], result["digest"], "reject",
                    reason=result["check"])       # e.g. "failed: /summary/flagged <= /summary/seen"
```

SECTION 09

Scope

AREA	WHAT SAPHAN STUDIO SUPPORTS
Frameworks	LangGraph, in Python; other agent frameworks run on the same generic flow runner
Model providers	one per flow, declared in the manifest; every call through the model gateway

AREA	WHAT SAPHAN STUDIO SUPPORTS
A flow that pauses for a human answer	the flow stops at a recorded checkpoint; the person's answer becomes a new signed order that continues from that checkpoint; nothing waits half-run
Manifest review	with every order, or registered once and reused by digest
Results for connectors	a notification when a result is ready; the connector reads the stored result
Direction B	from the machine where Saphan runs, or from another machine through the Saphan gateway
Editions	Team and Enterprise; Solo is not in scope

SECTION 10

Before your first flow

These answers decide how your first flows are set up. Send them to your Saphan contact or to hello@saphan.ai, quoting the number.

1. Which LangGraph and LangChain versions do you run? Do you need JavaScript as well as Python?
2. Which direction comes first for you: Saphan Studio calling your flow, or your application starting orders?
3. Who accepts results: a person, an automatic rule, or both? What must the stored result show for that decision?
4. Is Saphan's recorded session enough as the trace, or do you need third-party tracing, for example LangSmith, inside a governed run?
5. Which model provider does each of your first flows use?
6. Do your first flows pause for a human answer?
7. Should the signer review the manifest with every order, or register it once?
8. How large are your results and artifacts, and in which formats?
9. Who in your organisation should sign orders that your application prepares: anyone enrolled, or only people scoped to a flow and a budget?

SECTION 11

What this document does not claim

01 It describes what the boundary guarantees, not how confinement, signatures or the record are built. Saphan Studio's governance approach is U.S. and EU patent pending.

02 The limits it names are defaults; your installation can change them.

03 It makes no claim about LangGraph or LangChain beyond the versions a customer names.

SECTION 12

The Saphan ecosystem and this document set

Saphan Studio is one product in a small ecosystem. The tools around it are open source under Apache 2.0 and each works on its own; together they cover a change from the first read of the code to the evidence that it is right.

`Discover reads the code` → `Skill Studio checks the skills` →
`Saphan Studio runs the agents under a person's approval` → `Mock Studio stands in for real systems` →
`Test Studio and Proven judge the result`

TOOL	ITS PART
Saphan Studio saphan.ai	The control plane: signed orders, confined agents, independent review, human approval, one signed record.
Discover github.com/saphan-ai/discover-studio	Reads a codebase before anyone changes it, with a source for every claim; proposes the skills that fit it.
Skill Studio github.com/saphan-ai/skill-studio	Checks and signs the skills agents are given, so anyone can verify them again.
Mock Studio github.com/saphan-ai/mock-studio	Stand-in systems agents and tests work against instead of the real ones.
Test Studio github.com/saphan-ai/test-studio	Shows what a test run really proved, in the framework the team already uses.
Proven github.com/saphan-ai/proven	An AI code reviewer that reports a problem only with evidence.

The document set

DOCUMENT	WHAT IT ANSWERS
Saphan Studio for developers and DevOps	start here: what Saphan Studio is and how the pieces fit
Skills and craft	what every agent carries into the work
Testing and review	how a change earns its merge
Orchestration rules	a deterministic engine, model work inside it
Security	what an agent can reach, and what you can prove afterwards
Agent framework flows (this document)	how a customer's LangGraph flow runs signed, confined and recorded
Work across models	which model does which work, at what cost, reviewed by whom

For partners: a founding customer onboarding guide and an Enterprise continuity plan, at saphan.ai/for-partners.

All documents: saphan.ai/product · Code quality against published baselines: saphan.ai/research/proven-effort · Operator documentation: docs.saphan.ai · Contact: hello@saphan.ai