

HARNESSING THE AGENT: ***LAYERED VERIFICATION FOR AUTONOMOUS WORKFLOWS***

Vadim Vybornov

INTRO: WHAT IS AN AI AGENT?

Agent = LLM + tools + runtime

The LLM decides, the runtime executes authorized tool interactions.

LLM

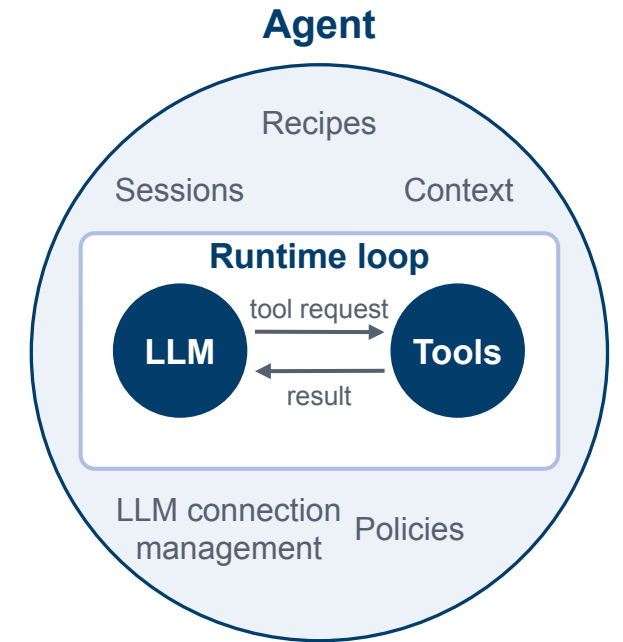
Interprets the request, reasons over available context and produces text or a request to use a tool. **It does not act.**

Runtime

- Manages the model↔tool interaction:
 - receives a tool request, executes an authorized call
 - runtime loop: returns the result to the model and repeats until the task is complete or a limit is reached
- Manages schemas, connections, timeouts.
- Manages the agentic role and sessions: role recipes, context, allowed tools, model providers, session history, etc.

Tools

- Local: shell commands, local MCP tools
- Remote: APIs via shell (curl, etc.), remote MCP services



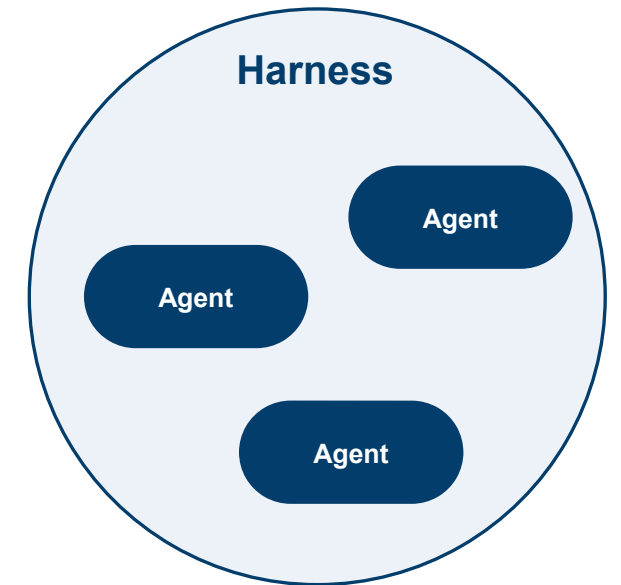
INTRO: WHAT IS AN AGENTIC HARNESS?

Harness = agents + control and operation level

The agent runtime lets an LLM use tools. The harness makes that execution governable, durable and operable.

Why is an agent runtime not enough? It does not guarantee:

Validation	Are outputs and actions correct, safe and complete? An agent can stop reasoning midway. Hallucinations and wrong decisions can corrupt valid workflows.
Durable workflows	Can the workflow survive retries, restarts, pauses and failures during long-running execution? A restart can trigger a non-idempotent action and submit duplicate jobs.
Tool authorization	Are the requested tool call, target and chosen parameters allowed at the given step? An injected wrong instruction can cause deletions or unauthorized actions.
Policy enforcement	Which environments, data, credentials and permissions may be used now? The agent can disclose secrets placed in its environment configuration.
Operational control	Can the system coordinate agents, manage and validate their I/O, provide observability and a human approval channel? Delegation may be ineffective: an agent refuses to delegate to a specific sub-agent, or does not wait for its response.



FROM FLEXIBLE REASONING TO CONTROLLED AUTONOMY

Strong reasoning alone is not enough for autonomous workflows.

1 Agents are not deterministic

They can focus on different things and the output is not the same for the same request.

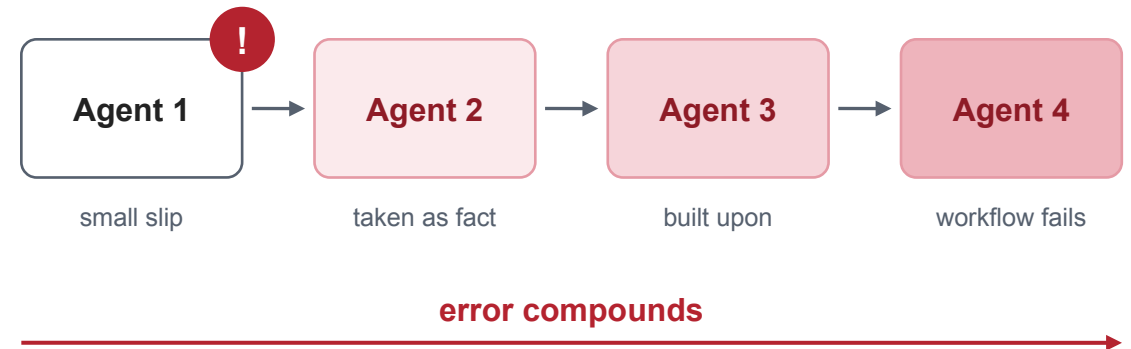
2 Every step influences the next one

An ambiguity can propagate, and the whole workflow may fail.

3 How can we verify the output automatically?

How to verify, evaluate and accept an autonomous agent's output, and propagate it further?

Every step influences the next one



Code checks. Model assesses. Human decides in critical cases.

CONSTRAIN INTERFACES, NOT REASONING

Every agent should have structured output. Why?

1

Enable autonomous workflows

Machine-readable output lets the system validate an agent's result automatically.

2

Better understanding for ourselves

What to expect from the agent in the context of our specific task. The LLM black box can hold far more information and knowledge than we can follow or possess.

3

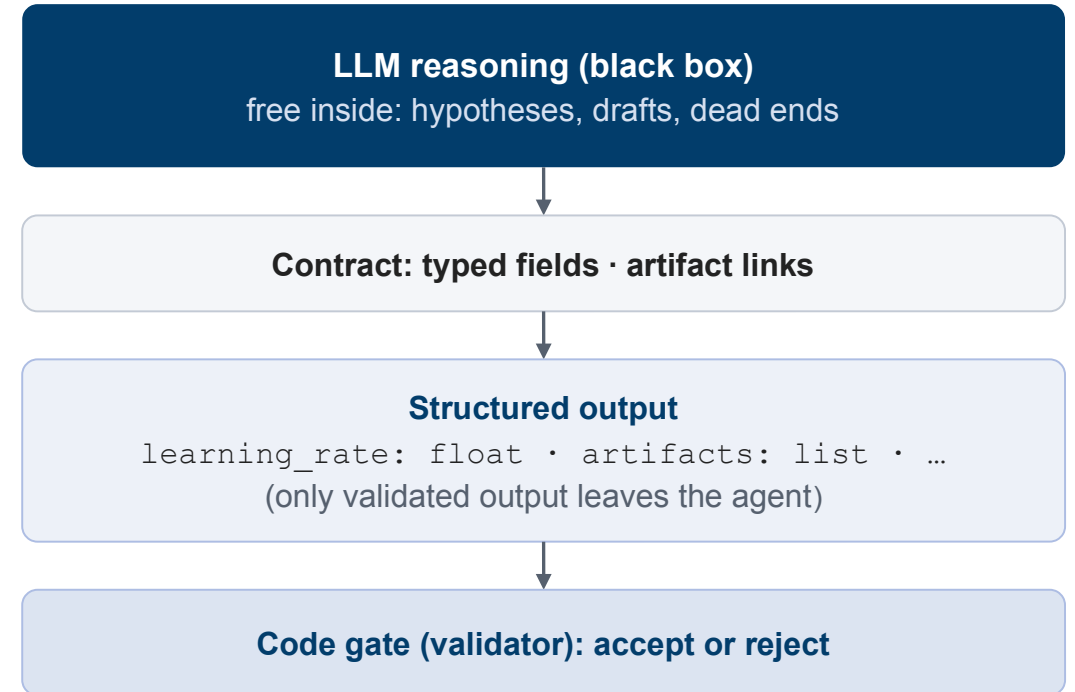
Enforce agent focus

We control what the LLM delivers, not what happens inside its reasoning black box.

4

Survive model drift

The model changes; the output – not.



Govern the output, not the reasoning

ANATOMY OF AN AGENT CONTRACT

The contract defines what the agent must deliver: fields, types and artifact references for verification.

Example of an output contract of the search agent

```
class HparamProposal(BaseModel):  
    learning_rate: float  
        ↳ required, float  
    batch_size: int  
        ↳ required, integer  
    image_size: int  
        ↳ required, integer  
    optimizer: str  
        ↳ required, name (e.g. adamw)  
    augmentations: list[str]  
        ↳ list of names, may be empty  
    hypothesis: str  
        ↳ required: why these values were chosen  
    artifacts: list[ArtifactRef]  
        ↳ file links: type, path, sha256 hash  
    reported_metrics: dict[str, float]  
        ↳ metrics the agent claims
```

Three possible agent responses

✗ REJECTED

Cannot be parsed into fields — nothing to verify

"I picked sensible parameters:
small learning rate, medium batch"

✗ REJECTED

No artifacts or reported_metrics — nothing to verify against

```
{"learning_rate": 0.0003, "batch_size": 16,  
 "image_size": 512, "optimizer": "adamw",  
 "augmentations": [...], "hypothesis": "..."}  
{"artifacts": [...], "reported_metrics": {...}}
```

✓ ACCEPTED

Structure is valid and verifiable → next step

```
{"learning_rate": 0.0003, "batch_size": 16,  
 "image_size": 512, "optimizer": "adamw",  
 "augmentations": ["flip", "color_jitter"],  
 "hypothesis": "A smaller step stabilizes loss",  
 "artifacts": [{"kind": "metrics",  
                  "uri": "runs/42/metrics.json", "sha256": "b7e2d0..."}],  
 "reported_metrics": {"val_miou": 0.74}}
```

An agent's output is always a serializable contract

VERIFY EXECUTION EVIDENCE, NOT AGENT CLAIMS

Contract checks: static guards, then evidence



Checking: learning_rate 0.0003 · batch_size 16 · image_size 512 · claimed val_miou 0.74 · file runs/42/metrics.json

Level 1. Static checks

Value bounds

learning_rate 0.0003 $\leq$ 0.1; batch_size 16 $\leq$ 64

✓ **passes** (0.5 would be rejected)

Memory estimate

batch 16 at 512×512: about 9 GB of 24 GB (estimate)

✓ **passes** (batch 64 would not fit)

Train/val leakage

shared ids between train and val: 0

✓ **passes** (a shared id would be rejected)

Level 2. Execution evidence

Run it and check claims against the contract's artifacts

File and hash

runs/42/metrics.json exists, sha256 b7e2d0... matches

✓ **passes**

Pilot run

1 epoch on 200 images: no OOM, no NaN in loss

✓ **passes**

Overfitting

train loss 0.15, val loss 0.18: gap within limits

✓ **passes**

Claimed metric

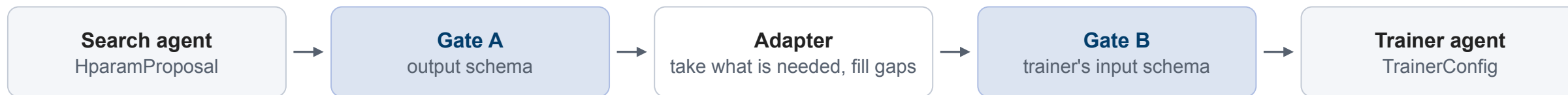
agent claimed val_miou 0.74, metrics.json says 0.71

✗ **REJECTED: claim not confirmed**

A claim is accepted only with an artifact verified without the LLM.

DECOUPLING AGENTS: VALIDATE → MAP → VALIDATE

Each agent has its own input schema. It decides what to accept; the adapter only assembles input from the previous agent's output.



Mismatch	Adapter action
<code>learning_rate → lr</code>	Rename the field
<code>seed and epochs missing</code>	Fill in 42 and 20 from config and write to the log
Extra: <code>hypothesis, artifacts, reported_metrics</code>	Not passed to the trainer: take only what is needed. They stay in the audit record.

Trainer input schema: allow-list

```
tools:          [train_segmentation]
datasets:       [urban_seg_v3]      # has masks
optimizer:      [adamw, sgd]
augmentations:  [flip, rotate, color_jitter]
lr:             1e-5 ... 0.1
batch_size:     1 ... 64
seed, epochs:   required
```

What the consumer's input schema rejects even if it exists

Wrong dataset

```
dataset: "urban_cls_v1"
```

✗ Exists, but has no masks and is not in the trainer's list

Wrong tool

```
tool: "train_classifier"
```

✗ The trainer may only use `train_segmentation`

Disallowed value

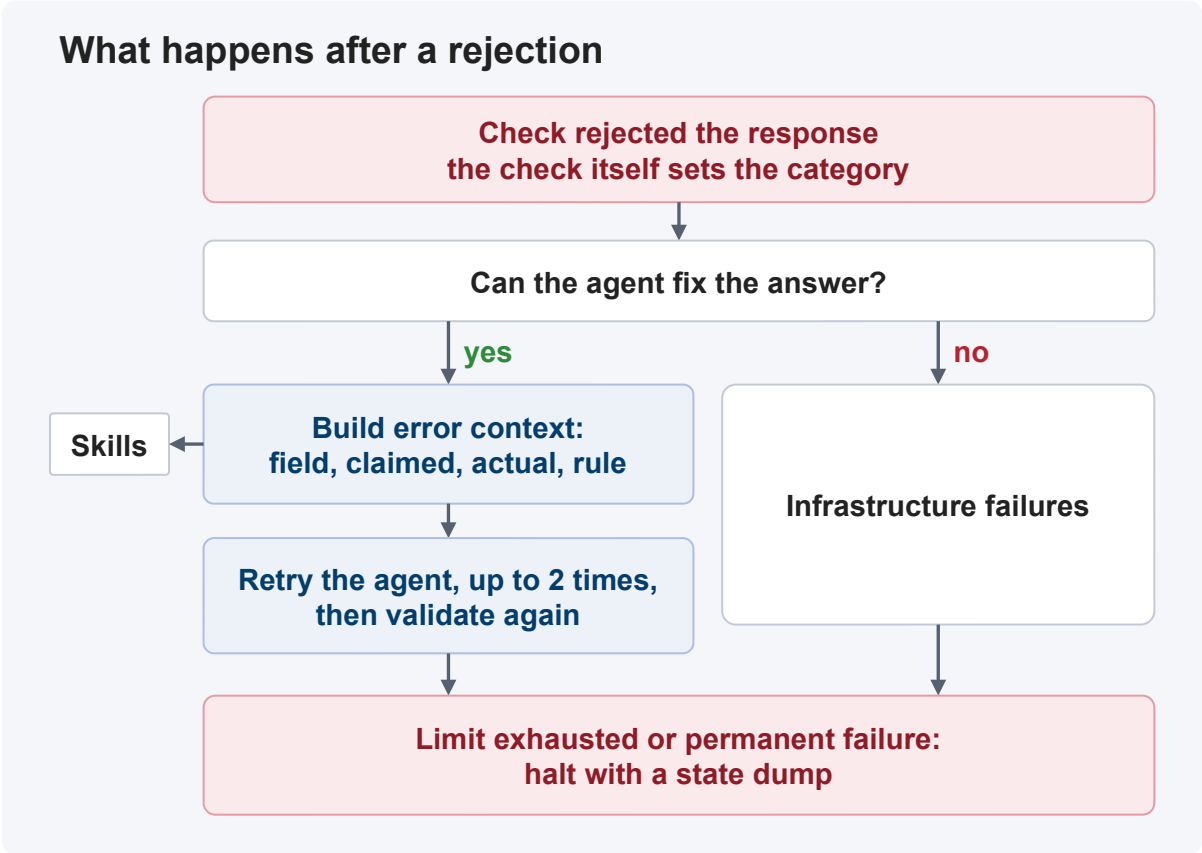
```
augmentations: ["mixup"]
```

✗ Not allowed for segmentation: it breaks masks

The agent proposes – the system decides

DETERMINISTIC FEEDBACK LOOPS: RETRY OR FAIL FAST

The error category picks the reaction: can the agent fix it by changing its answer? Retries are capped.



Error message

```
{ "status": "VALIDATION_ERROR",  
  "origin": "agent_output",  
  "field": "reported_metrics.val_miou",  
  "claimed": 0.74, "actual": 0.71,  
  "source": "runs/42/metrics.json",  
  "rule": "claimed must match actual" }
```

The error response schema is placed alongside the gate schema.

The agent sees what is wrong and what confirms it

Error	Category → reaction
Metric mismatch, value out of range, missing artifacts	agent → retry with error context
dataset_id not in registry	agent → retry with error context
dataset_id in registry, storage not responding	environment, transient → orchestrator retries itself
No GPU, no access rights	environment, permanent → stop at once

Every agent-related rejection is a lesson for the learning skills

FROM USER REQUEST TO CRITERIA TO VERDICT

Structure and numbers are valid, but the answer does not meet the request: the judge rejects it.

1. User request

"Propose the next experiment to improve segmentation. Use results from previous runs, pay attention to pedestrian quality, and explain why the chosen changes should help."

Verified facts

mIoU: 0.68 → 0.71

IoU pedestrian: 0.62 → 0.48

2. The judge builds criteria

"Use previous runs"

Does the proposal rely on specific results?

"Pay attention to pedestrians"

Does the rationale account for the pedestrian drop?

"Explain why it helps"

Is the parameter change linked to an expected effect?

Each criterion: PASS — met; FAIL — not met.

3. The judge rejects the answer

"mIoU rose from 0.68 to 0.71. Increase epochs from 20 to 30: longer training may raise overall mIoU."

Relies on runs: PASS

Accounts for pedestrian: FAIL

Explains effect: PASS

Result: FAIL — one required criterion is not met. Send the answer back for revision.

Reason: the pedestrian drop is not addressed; only the overall metric gain is explained.

The judge builds criteria from the request before its work, then applies them to the answer

WHAT IF THE JUDGE APPROVES THE WRONG ANSWER?

Suppose the judge approved based on wrong data.

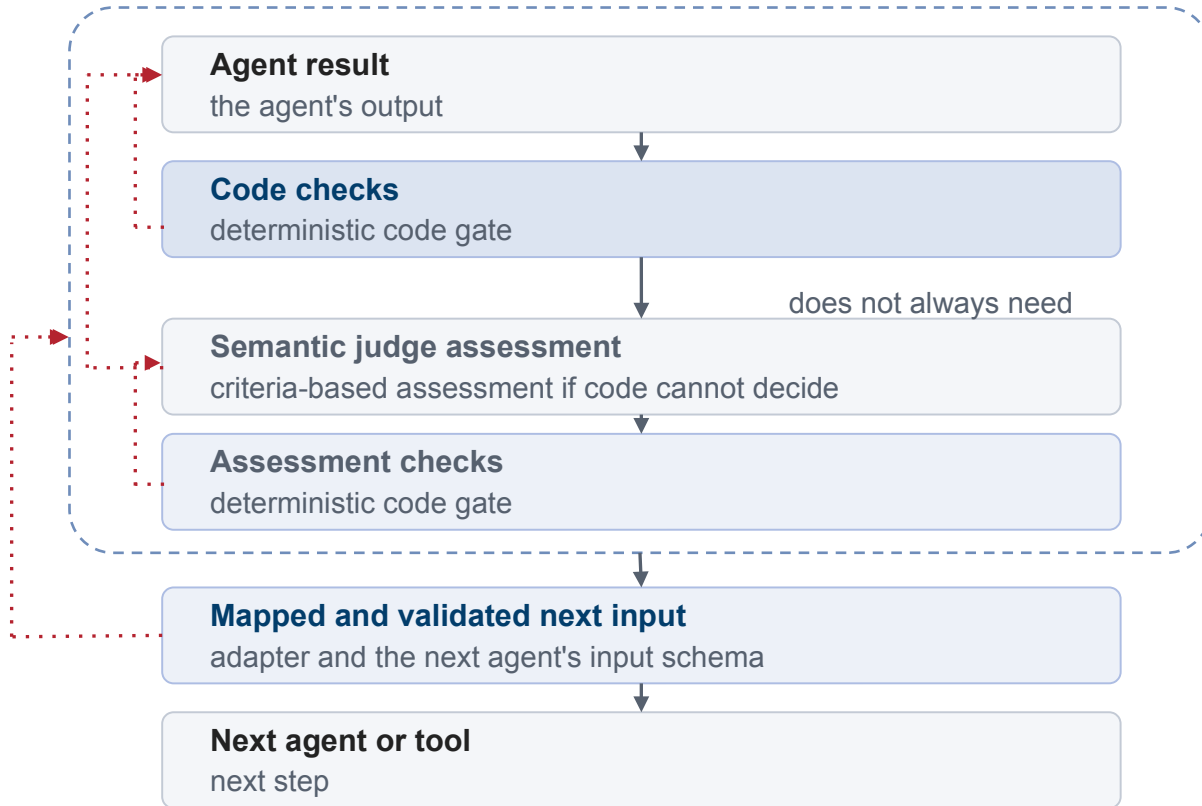
One template for all criteria; the judge fills in values for the request; code compares the record to the criterion field by field.

1. The judge passes the answer "mIoU rose from 0.68 to 0.71, IoU pedestrian: 0.93 → 0.95. Increase epochs from 20 to 30: longer training may raise overall mIoU." Relies on runs: PASS Accounts for pedestrian: PASS Explains effect: PASS	2. Criterion C2 from the request: "pedestrian quality" critterion_id: C2 subject: pedestrian evidence: class_metrics: <required> response_excerpt: <required> verdict: PASS FAIL For a different claim, the judge will create different criteria on the same form.	3. Judge's record filled in for this criterion critterion_id: C2 subject: road evidence: class_metrics: road 0.93→0.95 response_excerpt: "mIoU rose..." verdict: PASS	4. Code check field by field ✓ critterion_id matches ✗ subject: road ≠ pedestrian ✓ evidence filled in ✗ metrics not about subject ✓ quote found in answer ✓ verdict is valid
--	---	--	--

VALIDATION FAIL: subject: expected `pedestrian`, received `road`.
Verdict rejected by code. The judge gets the error with feedback and re-evaluates against the same criterion.

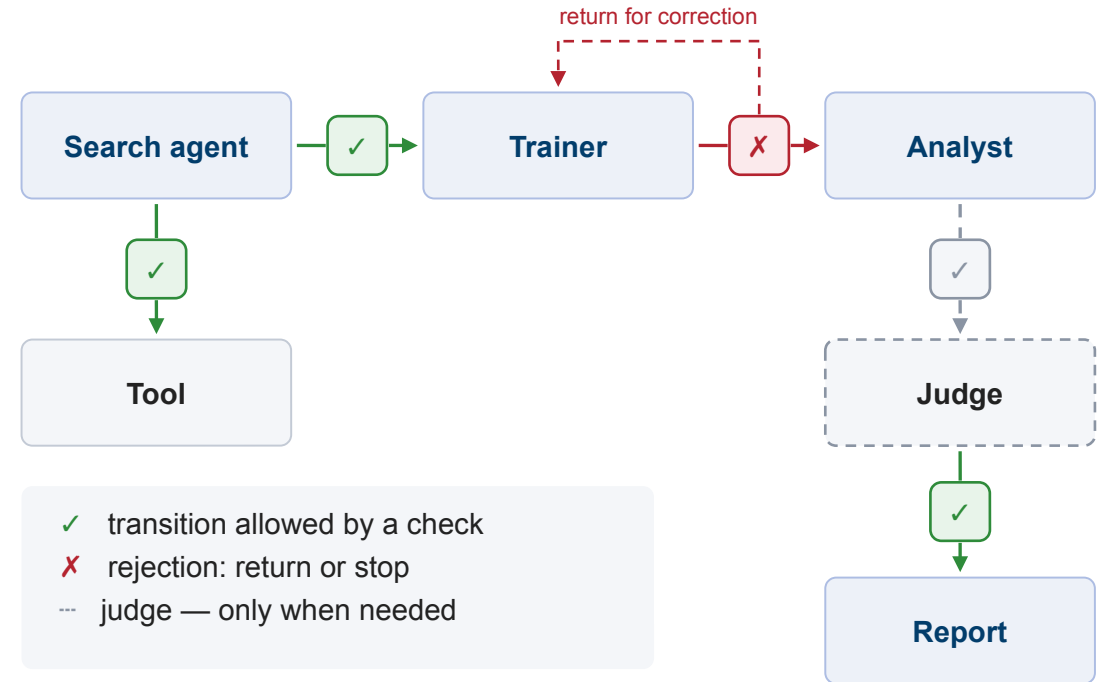
AGENT BUREAUCRACY ENABLES AGENT AUTONOMY

The executor submits a result, the assessor submits grounds, and rules decide the transition.



A rejection can occur at any step: fix or stop per policy.

Example pipeline



Transitions between agents are deterministic.

Making input-output at each step explicit, verifiable and correctable.

TAKEAWAY: CHECKLIST

What to check for an agentic pipeline.

Contracts

- ☐ Every agent has an input and output schema: fields, types, required artifacts.
- ☐ Do not put vague or hard-to-verify fields into the contract schema.
- ☐ Schemas are versioned; changes stay backward compatible.

Code gate

- ☐ Static checks: value bounds, memory, leakage between datasets.
- ☐ Verify claims against artifacts: files, hashes, metrics.
- ☐ Between agents: an adapter and the consumer's input schema with an allow-list.

Failures

- ☐ Every check has an error category: fixable by the agent, or an environment failure.
- ☐ Define retry limits.
- ☐ Define an error response schema: the agent receives error context.

Semantic judge

- ☐ Criteria are built from the request and fixed before assessment.
- ☐ The judge's record is checked by code: status, criterion, evidence.
- ☐ A PASS or FAIL without evidence for the right subject is not accepted.

Building block: agent with gates at boundaries

Thank you for your attention!