



AI AGENT DEVELOPMENT

PLAYBOOK 2026

FROM FIRST PROTOTYPE TO CONTROLLED DELIVERY

Architecture, tools, retrieval and memory

Approvals, evaluation, reliability and operations

Worked Python labs and reusable project worksheets

Read this first

This playbook takes you from a vague automation idea to an agent you can evaluate, constrain and operate. The running example is an internal support-triage assistant: it retrieves approved knowledge, drafts an answer and proposes an internal ticket note. A human reviews the exact note before the application writes it.

What you will build

The companion kit is a small, executable Python 3.11+ teaching harness. It includes tenant-filtered retrieval, strict decision validation, a bounded loop, payload-bound approval and a transactional, idempotent local write. Its model decisions are deterministic fixtures. It uses the standard library and makes no network calls. You can inspect every boundary before paying for a model API. The handbook then explains how to replace the fixture with a provider adapter, integrate a real ticket service, create an evaluation set and release gradually. Those extensions require your own authentication, credentials, infrastructure and acceptance testing. The kit is not a hosted service or a complete enterprise application.

Who this is for

You should be comfortable reading Python, JSON and HTTP requests. Technical leads can use the decision tables and worksheets without implementing the whole example. If you are new to Python, run the kit unchanged first and follow the glossary while reading.

How to use the package

- Start with the scope and architecture chapters. Write the acceptance contract before choosing a framework.
- Run the kit and its tests. Change one boundary at a time and observe which test fails.
- Complete the security, evaluation and deployment worksheets before connecting a system that holds real data.
- Keep the public-document reading trail separate from your private credentials, logs and customer records.

Edition and use

InformBytes, edition 1.0, 27 October 2026. Engineering guidance and original examples; not certification or a promise of business results. All organizations, tickets, policies and numerical examples are synthetic. Vendor documentation changes; check the linked official sources before implementation. No vendor affiliation or endorsement is implied.

A map of the engineering work

An agent project has two products: the behavior a user sees and the control system that makes that behavior dependable. Build them together. A fluent answer is weak evidence if the system accessed the wrong tenant or silently repeated a write.



Five artifacts before launch

The scope contract names the task and who may request it. The tool catalog lists the precise reads and writes. The evaluation pack describes outcomes and boundary failures. The release record ties a tested version to its configuration. The runbook tells an operator how to stop actions and reconcile uncertain results. These artifacts are intentionally small. A one-page contract used in every review is more useful than a long architecture document that never changes after the prototype. Put versions beside prompts, schemas, models, corpus snapshots and policy configuration so a failed run can be reproduced.

Separate proposals from authority

The model may propose a search, a draft or an action. Trusted application code decides whether that proposal is structurally valid, authorized, within budget and eligible to execute. Tool descriptions help the model choose; they do not grant permission. Retrieved documents provide evidence; they do not rewrite the application's rules.

First milestone

Take one synthetic request through the entire path: authenticated principal, authorized search, supported answer, explicit note proposal, exact-payload approval, one durable write and an audit record. Repeat it with another tenant and a changed approval payload. The second run should stop at the relevant boundary. A useful first demo therefore includes a successful case and a controlled failure. Record both. Do not infer product value, general accuracy or production safety from a single completed task.

Choose the smallest architecture

A deterministic program is often the best automation. If the inputs and branching rules are stable, ordinary code is easier to inspect. A language model becomes useful when interpreting varied language or synthesizing evidence adds value. An autonomous loop becomes useful only when the next useful step depends on what a previous step discovers.

01 CODE	Fixed inputs and branching
02 WORKFLOW	Known sequence, language tasks
03 BOUNDED AGENT	Next step depends on evidence
04 MULTI-AGENT	Independent work earns its overhead

Decision table

Situation	Start with	Evidence to add complexity
Known fields and fixed policy	Ordinary code	Unhandled language variation
One question, known sources	Retrieval plus one model call	Repeated missing context
Known review sequence	Explicit workflow	Paths truly vary by result
Dynamic research or diagnosis	Bounded agent loop	Measured gain over workflow
Independent specialized tasks	Coordinator and workers	Improved outcomes after overhead

Apply it to Northstar

A first implementation can retrieve a few approved passages and ask for a draft in one call. Use a loop only if the assistant must refine searches based on results. Ticket writing remains an explicit approval workflow even when retrieval is agentic. Different parts of one product can use different control patterns.

Framework selection criteria

Compare persistence, cancellation, observable tool calls, testability, schema validation, upgrade policy and deployment fit. Prototype the same task in the simplest implementation before adopting abstractions. Measure the operational cost of the framework as well as model cost. S1 distinguishes predefined workflows from agents whose models direct their next steps. This handbook uses that distinction as a design aid. The architecture choice still depends on your task and evidence, not on the label a framework uses.

Decision record

Record the alternatives, chosen boundary, rejected complexity and a measurable trigger for revisiting the choice. 'We need more agents' is not a trigger. 'A single bounded loop repeatedly misses independent document sets that can be searched in parallel' is testable.

Draw the trust boundaries

A system diagram should show where authority changes, not only where data moves. A model, a retrieved document and a remote tool server are separate trust domains. A box named 'agent' hides too much to review.



Request path

The gateway authenticates the user and establishes the tenant. The orchestrator loads the versioned task contract, remaining budget and allowed tool registry. A model adapter returns a proposed decision. Validation and policy checks run before a tool dispatcher receives any arguments. Tool results become bounded evidence for the next model call. The approval service receives a canonical action and issues a short-lived approval record through a trusted reviewer interface. The writer rechecks identity, target authorization, digest and expiry immediately before the side effect. The model cannot approve itself.

Evidence path

Retrieval performs access filtering before ranking. Every returned passage carries an identifier, source version and enough provenance for later inspection. The response validator checks citation membership and output shape. A separate quality check determines whether the cited passage actually supports the claim.

Operational path

An audit event links request, tool invocation, approval and write result. A release identifier ties the run to its prompt, model configuration and tool schema. An operator can disable writes without breaking read-only drafts. Long-running jobs have cancellation and recovery paths.

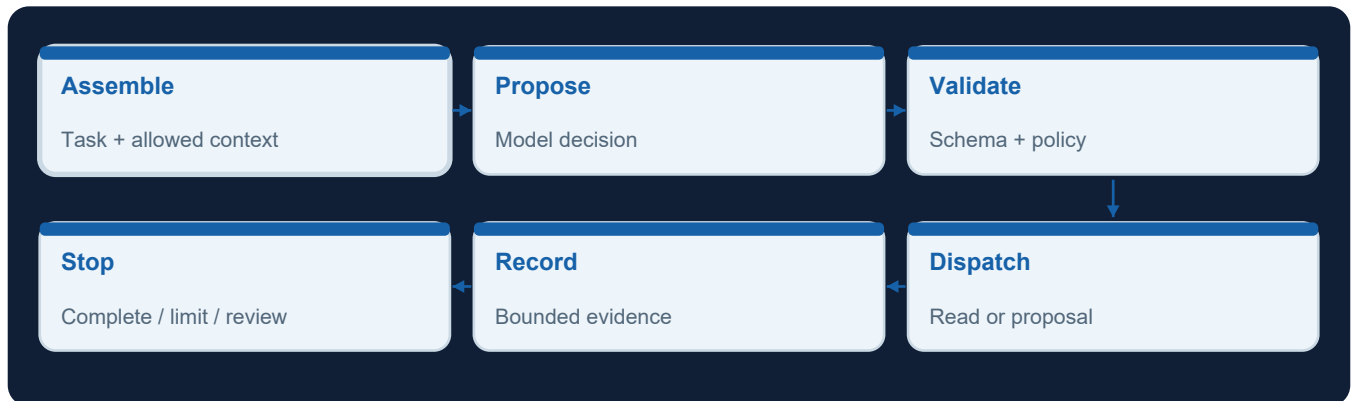
Review questions

- Which component establishes the tenant, and can model arguments override it?
- Which tool can create irreversible effects, and where is its policy enforced?
- Can an external document reach a privileged instruction channel?
- After a crash, can an operator distinguish an unapplied action from a completed one?

Answer these with named components and stored fields. 'The prompt tells it to be safe' does not identify an enforcement point.

Build a bounded decision loop

A loop needs clear exits, resource accounting and a record of what happened. 'Keep trying until successful' is not a usable execution policy for a system that can consume money or call tools.



The decision cycle

The application assembles allowed context and calls the model adapter. It validates the decision, rejects unknown fields, checks the budget and then dispatches a permitted read or creates a proposal. The next call receives the bounded result. A final answer must satisfy the output contract before it reaches the user.

Enforce several budgets

A step limit bounds the number of decisions. A wall-clock deadline bounds elapsed processing. Provider timeouts bound individual network calls. Token and spend reservations bound resource use before each call. Result-size limits bound memory and context growth. These controls address different failure modes and should not be collapsed into one setting. The offline kit implements steps, repeated-decision detection and elapsed-time checks. It does not implement a provider token meter or interrupt a blocked network call. A real adapter needs its own connect/read timeout and cancellation behavior.

Stop with a useful status

Return COMPLETE, NEEDS_APPROVAL, NEEDS_INPUT, LIMIT_REACHED, TOOL_UNAVAILABLE or OUTCOME_UNKNOWN as explicit states. Do not turn a limit into a fabricated answer. Preserve completed evidence and explain the next safe step.

Repetition policy

The kit treats an identical decision as a stop condition. A production search may legitimately repeat after a corpus update or tool recovery. Include the relevant tool version and attempt context in the fingerprint, and allow only deliberate, bounded retries. Do not disable repetition detection merely because the first task that hits it looks inconvenient.

Run the reference kit

The companion ZIP contains runtime.py, test_runtime.py, README.md and editable worksheets. Extract it into a new local folder. The kit uses only the Python standard library. It does not install packages, contact a model provider or connect to your ticketing system.

```
python --version
python runtime.py
python -m unittest -v test_runtime.py
```

Use Python 3.11 or later. On systems where Python is named python3, substitute that command. The demo should return status complete and the citation kb-a-1. The test command should report 22 passing tests in the supplied edition.

What the demo does

The fixture first proposes a search for Reset. The read function filters synthetic documents to tenant alpha and returns a matching passage. The next fixture decision returns the reset guidance with that passage's ID. The loop verifies that the citation was actually retrieved in this run. This validates deterministic boundaries and wiring. It does not measure a real model's ability to follow instructions, retrieve the right passage or produce a supported answer. Add model trials only after you can reproduce the offline behavior.

Try three changes

Change the citation to kb-b-1: the answer should be rejected. Add a tenant key to a search decision: strict validation should reject it. Change the fixture to propose a ticket note: the runtime should return needs_approval while the notes table remains empty.

Read the tests before extending

The approval tests cover changed payload, missing or expired approval, single-use consumption and replay conflicts. The persistence test closes one connection and reopens the same SQLite file. Treat these as executable explanations of the intended behavior.

Troubleshooting

If SQLite cannot create a file, use a writable local directory and inspect filesystem permissions. Avoid network-mounted database files for this exercise. If the demo returns no matches after you edit the document, inspect tokenization; the teaching search is word overlap, not semantic retrieval.