BUILDING PRACTICAL AI WORKFLOWS

A getting-started guide to preserving context, keeping people in control and turning requests into useful results

THE APPROACH

A safe agent system is not one giant model. It is a set of small, understandable parts: a front door, a worker that owns the loop, one guarded way to read files and a durable place for results.

Better continuity for consequential human work.

The work between the moments is where clarity gets lost. This guide shows one practical way to preserve context, keep humans in control and make the system's boundaries visible.

For leaders

A concise view of the operating model, the control boundaries and the decisions worth making.

For engineers

A step-by-step path, interface guidance, test checklist and starter prompt for a first monorepo scaffold.

Start here

The simple idea

You do not need a giant platform to start. You need one small request, a few tools, one safe way to read files and a saved result you can inspect.

Ask

Someone describes a task in plain language.

Check

The system decides what is allowed before it reads or acts.

Return

The person gets a useful result they can review and reuse.

Why split it into parts?

When each part has one job, you can test it by itself. That makes the system easier to learn, safer to change and less mysterious when something goes wrong.

The four rules that keep the pattern clear

01	One safe file door	Every worker uses the same path checker. No shortcut around it.
02	Thin front door	The gateway maps, validates and reports. The agent loop lives elsewhere.
03	Worker owns the loop	Planning, tools, progress and completion stay in the runner.
04	Results are durable	Drafts and scaffolds return as text and saved artifacts, not hidden file edits.

What is fixed, what is a choice and what comes later?

Fixed for this pattern

Every file read passes the safety check. The runner owns the work loop. Outputs return for review. People set goals.

Your choice

Python or TypeScript. npm or pnpm. Local folder or object storage. Direct call or queue.

Later concerns

Authentication, retention, tenancy, rollback, monitoring and cost controls.

A simple handoff rule

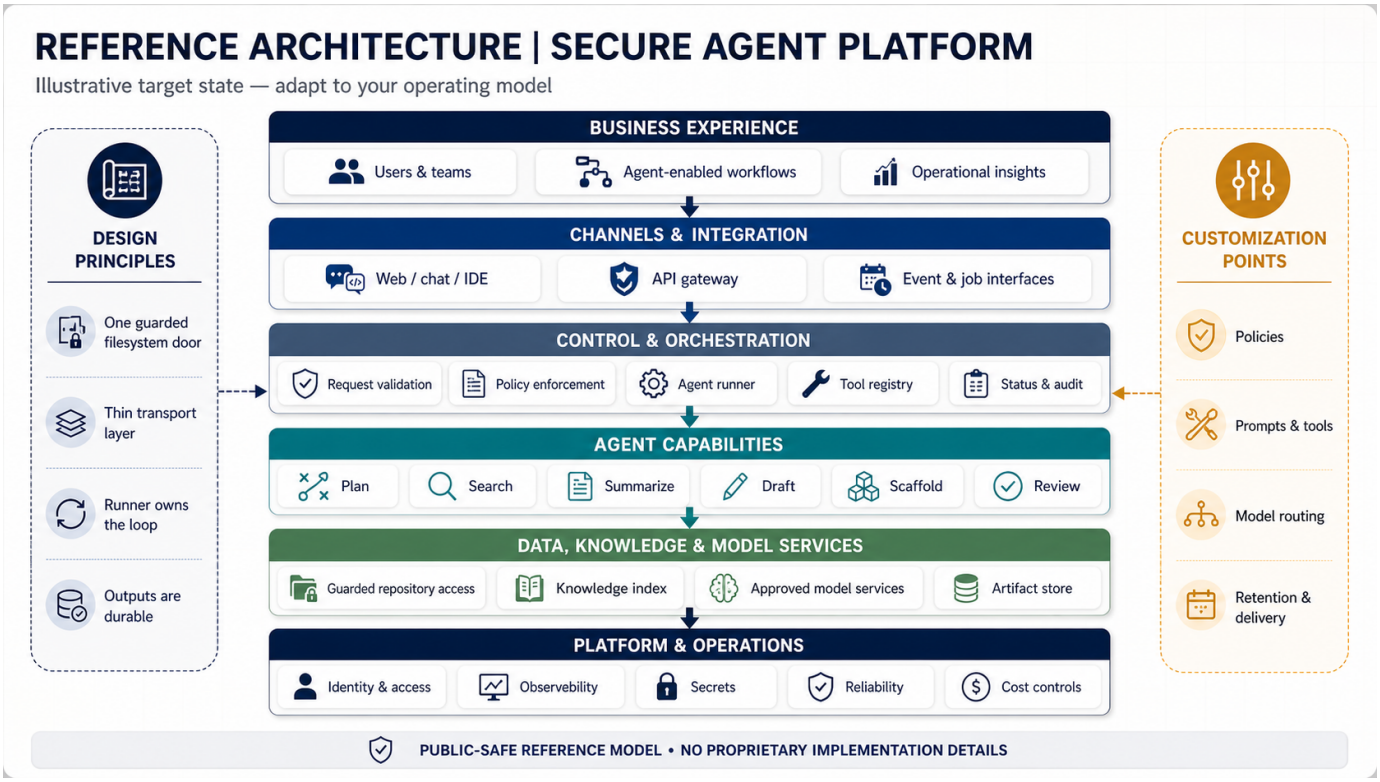
Start with a direct call when the task is short and predictable. Move to a job queue when the work may run longer than a few seconds, has several steps, needs progress updates or should continue if the client disconnects.

How to use this guide

Treat each section as a small lesson. Try the smallest version, run the checks, then add one useful capability at a time. The goal is to help you build understanding through practice.

1. Architecture map

Use this as a map of the pieces. You do not need to build every box on day one.

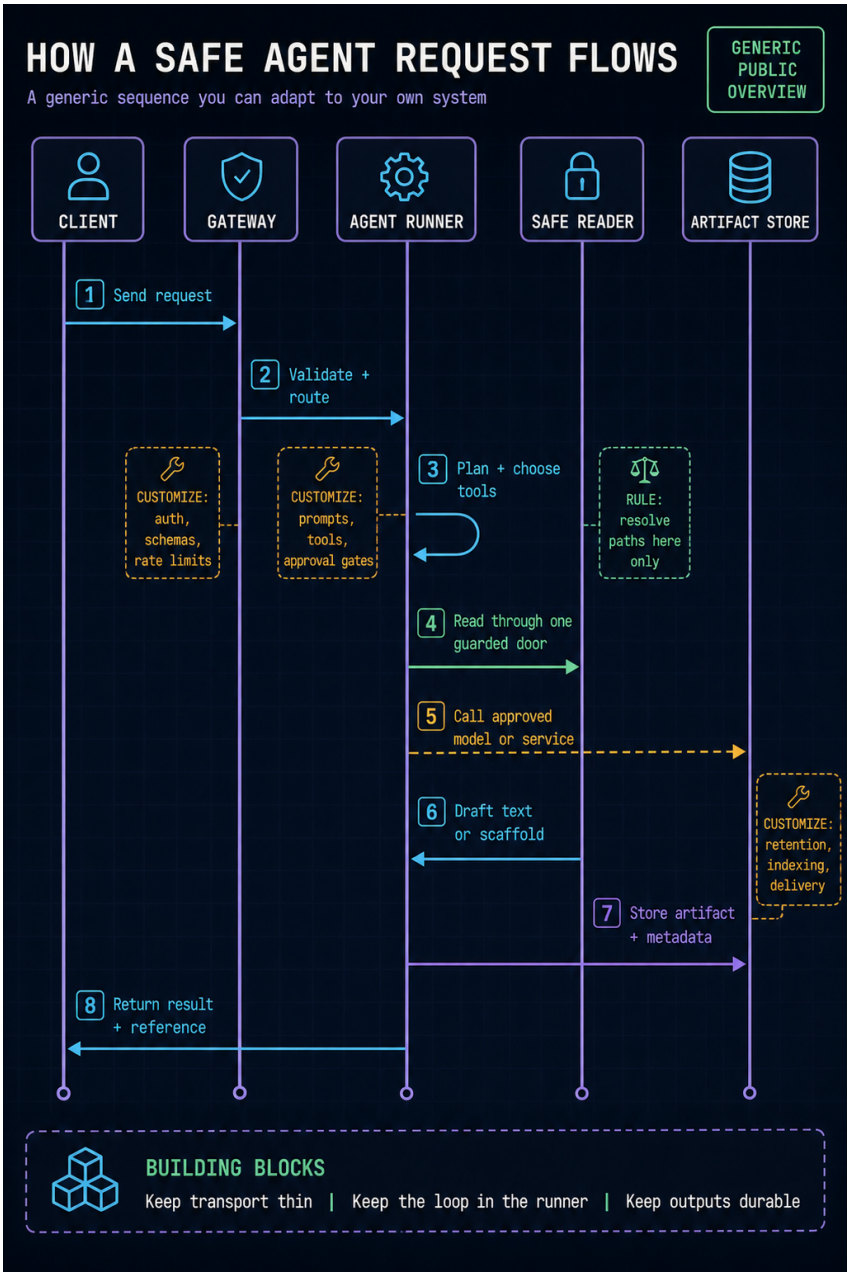


Start with the front door, the worker, the safe reader and the saved result. Add the rest when a real need appears.

Path safety is deterministic code. Resolve the requested path then allow it only when it stays under the approved root. A request such as ../secret is rejected before the model or tool reads it.

2. Sequence: one request through the system

Use this view to explain what happens in time order, from request to saved result.



Think of the gateway as the receptionist. It accepts the request, checks basic rules and gives the work an ID. The runner is the worker. It uses the ID, does the work and saves the result. A local version can call the runner directly. A deployed version may use a queue.

3. Start with a small loop

The first version can be simple. Make one useful path work before you add more tools or more automation.

Ask

A person describes a small task in plain language.

Check

The system checks the request and the allowed file boundary.

Work

The runner uses one or two tools and reports what it is doing.

Save

The result is returned as text and saved for review.

A good first project

Ask the system to read a harmless sample folder, summarize it and save the summary. That single path teaches you about requests, safety, tools, outputs and evals without requiring a giant platform. An artifact is like a separate draft report saved for review, not permission for the AI to edit the master repository.

Level up your knowledge and skill

Once the loop works, change one thing at a time. Add a tool, add an eval or add a better output format. Run the same example again so you can see what changed.

4. Blueprint: what each part does

Build one small path from request to result. Think of the system like a small team: the gateway receives the request, the runner does the work, safety checks file paths, tools do single jobs and artifacts save the result.

Suggested module boundaries

A product manager can use this page to ask who owns each decision. A junior engineer can use it to decide where code belongs and what to test first.

PART	JOB	EXAMPLE	CHECK
Gateway	Receives a request and reports status.	`submit(request) -> job_id`	Can I send a request and see its status?
Runner	Chooses the next step and calls tools.	`run(job) -> result`	Does it stop when the work is done?
Safety	Checks every path before a file is read.	`safe_resolve(root, path)`	Does ../ fail while a safe path works?
Tools	Do one clear task and return structured data.	`read(file) -> text`	Can another tool use the result?
Artifacts	Saves the output and returns a reference.	`store(text) -> ref`	Can I retrieve it later?

Build in this order

- Make the gateway return a health check.
- Make the safety check accept one safe sample path and reject one unsafe path.
- Make one tool read the sample.
- Make the runner call that tool.
- Save the result and return its reference.

Safe reader example

The model is not the security boundary. The code checks that the resolved path stays under the approved root. If it does not, the read stops before the tool runs.

Your first win

Build one request that reads a harmless sample folder, summarizes it, saves the summary and returns the saved reference. When that works, you already have a real system to improve.

5. Make progress in small wins

Each step gives you something you can run, check and show.

#	STEP	DO
01	Define the request	Write the input, allowed actions, output shape, errors and artifact reference.
02	Build the gateway	Add health, submit, status and result operations. Keep the gateway free of agent planning.
03	Build the safe reader	Allow selected roots, reject traversal and enforce read-only access.
04	Build the runner	Plan, choose tools, report progress and stop on completion or approval.
05	Add five small tools	Start with scan, find, read, summarize and draft.
06	Save the result	Store the text and metadata; return a reference that another person can retrieve.
07	Test the boundary	Prove allowed paths work and unsafe paths fail; repeat the demo with sample data.

When a run fails

Record tool inputs and outputs separately from model responses. Use the same request ID for both and redact secrets. This helps you tell a code or tool failure from a reasoning failure.

Your first four rewards

YOU DO	YOU CHECK	YOU GET
Run the health check	It returns a clear response	A system you can see
Read one allowed folder	An unsafe path is rejected	A real safety boundary
Run one tool	The result is structured	A reusable building block
Save one draft	You can retrieve it later	A useful end-to-end loop

Level up your knowledge and skill with evals!

When you add a new boundary, add one small eval beside it. The habit matters more than building a giant test suite on day one.

What counts as done

A health check, a blocked unsafe path, one synthetic end-to-end request, one saved artifact and a README another person can follow. That is enough for a first version.

6. What to add after the first version

Mature the system in layers

This roadmap is a menu, not a required sequence. Reorder it based on your users, data, risk and deployment context.

First local version

Safe paths, read-only access, one runner, a few tools, saved artifacts, health and repeatable tests.

Small deployed version

Authentication, approvals, audit events, timeouts, retries, usage limits, better search and a real artifact backend.

Production concerns

Queues, multiple workers, policy-as-code, evaluation suites, tenancy, rollback, monitoring, cost controls and deployment automation.

Decision questions for a real deployment

Who can ask the system to do work?	What may it read?
Which actions need approval?	Where do artifacts live and how long?
How are failures and model costs seen?	What record should be kept for review?

Use these questions when you adapt the pattern

The answers will change from team to team. That is the point: the pattern gives you a place to start then your context decides what comes next.

7. Where evals fit

Most useful systems have several eval layers. Each layer answers a different question and should change only the decisions it is responsible for. You do not need every layer on day one. Add the layer when the system reaches that stage.

Two levels are useful here. Model and tool evals live inside a specific capability and can test groundedness, hallucinations, retrieval quality, citations, tone or instruction following. The five rows below are system-level evals: they check whether the full workflow is built safely, remains useful and stays within approved boundaries.

For every eval, name the owner, the automatic check and the human decision. The agent should not quietly invent the goal or change the rules.

LAYER	SITS AT	QUESTIONS	EVAL EXAMPLE
Build	Code and contracts	Can it start, finish and return the expected information?	Owner: engineer. Automatic: run empty and normal requests. Pass: clear error, valid result and ready health check. Human reviews failures.
QA	Tools and boundaries	Do allowed actions work and do blocked actions stay blocked?	Owner: engineer plus boundary owner. Automatic: test safe and unsafe paths plus a missing service. Pass: safe works and unsafe rejects. A person approves boundary changes.
Product input	After real use	Does this help someone complete the task with less confusion or rework?	Owner: product manager plus users. User feedback: automate a 1–5 survey and collect comments after the task. System signals: track completion, retries, edits and abandonment. Pass: use both sources to decide what to improve.

7. Where evals fit, continued

The remaining system-level layers cover outcome ownership and controlled change. They make it clear who sets the goal, who checks it and what the system may adjust.

LAYER	SITS AT	QUESTIONS	EVAL EXAMPLE
Goal setting	Start and end of a work cycle	Did we deliver the outcome we agreed on?	Owner: product manager plus team before work starts. Goal: summarize a sample folder and include 3 required facts. Automatic: check all 3 facts are present, source references are included and the summary is saved. Human: confirm the facts are accurate.
Governance	Approved adaptation	What may the system change safely and what needs a person?	Owner: policy owner. Config: allow wording or approved tool order only. Pass: change stays on the list. A person approves access, permission or goal changes.

The demarcation points

- Build evals protect the code.
- QA evals protect the boundaries.
- Product feedback protects usefulness.
- Goal evals protect direction.
- Governance and adaptation stay inside the limits a person approves.

8. Starter prompt for a custom monorepo

Copy the complete prompt below into a coding agent. The headings are labels inside one prompt so you can read, edit and review it without losing the overall instruction.

ONE COMPLETE PROMPT

```
1. Context and goal:
Build a small monorepo for a safe AI agent. A monorepo means several related parts
kept in one project. Use native npm workspaces for this starter so apps can use shared
packages from the same project. pnpm is a valid later choice. Treat this as a
getting-started scaffold, not a finished product or the only correct architecture.

A person sends a request. A front door checks it. A separate agent worker chooses safe
tools. All file reads go through one safety checker. The system returns a draft or
scaffold as text and also saves a copy as a result. An artifact is a separate draft
report for review, not a hidden edit to the master repository.

2. Suggested folders:
apps/gateway: API for health, requests, status and results
apps/runner: agent loop, planning, tool choice and progress
packages/contracts: shared request, response, event and artifact shapes
packages/tools: scan, find, read, summarize and draft
packages/safety: safe path checks and read-only rules
packages/artifacts: save results behind a small interface
packages/config: settings with safe defaults; never commit secrets
tests: safety tests and one full request-flow test
docs: setup, architecture, sequence, safety notes and examples

3. Safety rules:
1. Every file read must use the safety package. No shortcut is allowed. The model is
not the security boundary.
2. Repository access is read-only by default.
3. The gateway only checks and routes requests. It does not run the agent loop.
4. The runner owns planning, tool use, progress and completion.
5. Draft and scaffold tools return text for a person to review and apply. Do not
silently edit the source repository.
6. Save outputs with simple metadata and a reference.
7. Show clear errors for bad paths, denied actions, timeouts and unavailable services.
Retry only temporary external service failures up to 3 times with exponential backoff
and a maximum time limit. Do not retry unsafe paths, denied actions or invalid
requests.
8. If an automatic adjustment causes two evaluation failures in a row, restore the
previous version, stop the adjustment and ask for human review.
```

8. Starter prompt, continued

Continue the same prompt. These sections describe the first demo, deliverables, evaluation plan and working method.

ONE COMPLETE PROMPT, CONTINUED

```
4. First demo:
Read a safe sample folder, summarize it, save the summary and return the saved result
reference. Use harmless sample data so another engineer can run the example safely.

5. Deliver:
- local setup that runs
- one fake end-to-end example
- tests showing safe paths work and unsafe paths fail
- health check
- short README written for a new developer
- adapters so model, search and storage can be swapped later

6. Evaluation plan:
Model and tool evals may test groundedness, hallucinations, retrieval quality,
citations, tone and instruction following inside a specific capability. System-level
evals should name the owner, the automatic check and the human decision. Include build
checks for code and contracts; QA checks for tools, paths and edge cases; user
feedback plus system signals after people use the result; goal checks against an
outcome set by the product manager and team before work starts; and governance and
adaptation only where the allowed change is explicit. If two evaluation runs fail
after an adjustment, roll back to the previous version and ask for human review.

7. Working method:
Before writing code, show the proposed file tree, the assumptions and the first
end-to-end test. Then build the smallest working version and run the tests. Explain
which choices are examples, which boundaries are fixed and which production concerns
come later.
```

About Ascendvent

Better continuity for consequential human work.

Ryan K. McDonald

Founder of Ascendvent.

Throughout my career I have watched organizations lose context between moments of action, forcing professionals to rebuild understanding instead of building momentum.

Ascendvent focuses on the guidance infrastructure that helps people keep context, make better decisions and stay at the center of the work.

Visit ascendvent.life

Use this guide to start a conversation

If your team is exploring agent workflows, secure repository access, internal developer tools or a practical AI operating model, this reference can be adapted into a working discovery and build plan.

ASCENDVENT | ASCEND AND REINVENT