AMODAL CLOUD

Platform & Runtime Documentation

Define an agent in versioned source files, enforce deterministic guardrails and policies, evaluate across models, and deploy governed agents and custom runtime applications on Amodal.

Contents

OVERVIEW

Amodal Cloud

START

Build With A Coding Agent
Project Structure
Run Locally (CLI)
Build A Support Agent
Source & Deploy In Amodal

AGENT REPO

amodal.json
Agents & Subagents
Connections
OAuth Connections
Skills
Knowledge Base
Stores
Tools
Agent Workflows
Hooks
MCP Servers
Runtime Apps
Evals
Confirmation Model
Guardrails

PLATFORM

Amodal Configuration
Issues
Proposals
Providers
Authentication
Embedding & Multi-tenancy
Deployment API

PLATFORM API

Platform API Reference

REACT SDK

SDK Overview
@amodalai/react
Chat Widget

REFERENCE

Runtime Server
Event Bus

OVERVIEW

Amodal Cloud

Amodal is the platform and runtime for building, governing, evaluating, and deploying production AI agents and agent-infused applications.

Every Amodal agent is defined as versioned source code starting at `amodal.json`: instructions, API and MCP connections, domain knowledge, typed data stores, custom TypeScript tools, deterministic or LLM-judged guardrails, evaluation suites, and optional custom React operator interfaces. Source lives in a connected GitHub or GitLab repository or in Amodal-hosted source versions. You can author and evolve an agent directly in the Amodal Studio UI (using visual tabs or the built-in **Builder Chat** assistant that drafts changes as reviewable Proposals), work locally in Git with your IDE or coding agent, or combine both with two-way repository synchronization.

What You Can Do With Amodal

- **Connect to existing systems and actuate workflows (read and write):** Most enterprise workflows span existing databases, REST APIs, SaaS systems, and internal services. With Connections, OAuth Connections, and MCP Servers, an agent can inspect schemas, query live records, and execute write operations (updating case files, creating payment plans, triggering operational playbooks, or filing records) under explicit access rules and human-in-the-loop approvals (Confirmation Model).
- **Enforce deterministic guardrails, PII redaction, and scoped access:** Place programmable Guardrails and Hooks directly in the execution path between the user, the model, and downstream APIs. Strip sensitive fields at the connection boundary (`policy.json`), redact PII such as Social Security numbers with deterministic regexes or external services before data reaches the model or the user, block out-of-policy tool calls (for example, rejecting payment plans above a dollar threshold), and isolate sessions, memory, stores, and credentials per user or tenant using cryptographic Scopes.
- **Ground agents in large domain corpora and structured stores:** Combine always-on operating rules with hybrid-indexed Knowledge Base corpora (supporting thousands of pages of manuals, rulings, regulations, and technical specifications with section and line citations) and runtime knowledge libraries where operators file documents on the fly. Pair unstructured reference material with typed Stores (Postgres, SQLite, or S3) for structured records shared between the agent and your application.
- **Stay model-agnostic and benchmark models with Evals:** Run Amodal's agentic reasoning loop on top of hosted frontier models, bring-your-own-key (BYOK) provider credentials, or private on-premises and air-gapped open-weight models (Providers). Use Evals as continuous integration for your agent, combining deterministic tool-call assertions with LLM-as-a-judge rubrics to run multi-model bake-offs comparing accuracy, latency, and token cost before promoting a deployment.
- **Deploy standalone chat, workspace channels, or full custom operator applications:** Every deployed agent immediately serves a hosted chat interface and streaming Runtime Server, and can be connected to channels such as Slack (Amodal Configuration). When a workflow requires more than a chat box, build a custom React Runtime App or embed `@amodalai/react` into an existing web application so the agent works alongside dashboards, tables, evidence viewers, and queues—reading the operator's screen context (`scope_context`) and streaming live state updates back to the UI.

- **Close the loop with telemetry, online evaluators, issues, and proposals:** Every turn is captured in OpenTelemetry Traces and Sessions. Online **Trace Evaluators** continuously grade live human-agent interactions in production, while users can flag bad turns with thumbs-down **Feedback**. Failed evaluations and user feedback automatically group into Issues, where Amodal can diagnose the root cause against the session trace and generate a Git-backed Proposal containing the source fix and regression eval—ready for review, approval, production promotion, or instant rollback.

How The Platform Fits Together

Stage	What happens	Key surfaces
1. Author & Connect	Start from a template, blank agent, or Git repo. Add instructions, REST/MCP connections, knowledge, stores, and tools via Git or Studio's **Builder Chat**.	`amodal.json`, Connections, Proposals
2. Guard & Ground	Restrict API surfaces with `policy.json`, attach pre/post turn and tool hooks for PII redaction or policy checks, and require human confirmation on high-impact actions.	Guardrails, Hooks, Confirmation Model
3. Evaluate & Select	Run deterministic and judge-based eval suites across candidate models to verify behavior, compare latency and cost, and gate releases.	Evals, Providers
4. Deploy & Embed	Build immutable source versions, test preview deploy URLs, promote to production, and expose the agent via hosted chat, custom React runtime apps, or embedded SDKs.	Create & Deploy, Runtime Apps, React SDK
5. Observe & Improve	Inspect live sessions, traces, and logs; run online trace evaluators; triage failed turns or user feedback into issues; and approve AI-drafted fix proposals.	Amodal Configuration, Issues, Proposals

Build an agent

1. Build with a coding agent: give your coding agent the setup prompt.
2. Project Structure: where each file belongs.
3. Run locally to try an agent on your machine.
4. Build a support agent: a complete example against a public demo API.
5. Source & deploy in Amodal: create, deploy, and select production in the app.

Core concepts

Concept	Meaning	Example
Agent	A deployed definition of instructions and capabilities.	`support-agent`
Session	One conversation, including messages, tool calls, approvals, and usage.	A ticket investigation
Scope	An identifier that labels sessions and partitions memory, non-shared runtime stores, and scoped credentials.	`tenant:acme`

Concept	Meaning	Example
Scope context	Facts the application supplies for the current turn.	`{ "ticket_id": "123", "page": "support" }`
Named agent	A prompt and a selected set of capabilities under `agents/<name>/`.	`review`

Use `scope_id` to identify the application boundary and `session_id` to continue a conversation. Several sessions can share a scope. Sending the same `scope_id` without its `session_id` starts a separate conversation.

Scope context is information for the model, not an authorization claim. A runtime token minted with a `scope_id` confines chat and session history to that scope, so your backend must mint each token only for a scope the user may access. Tools and connections must also enforce tenant access; see Embedding & Multi-tenancy.

Deploy, test, and embed

- Evals: check behavior before deploying changes.
- Providers: choose models and configure provider access.
- Authentication: control access to the hosted app and runtime.
- Deployment API: deploy, activate, promote, and roll back.
- React SDK: embed chat and agent data in your application.
- Runtime Apps: deploy a custom frontend at the agent's URL.
- Runtime Server: call chat endpoints and consume streaming responses.
- Download PDF Documentation: offline PDF bundle of the complete Amodal documentation.

START

Build With A Coding Agent

Paste this into your coding agent:

```
Fetch this setup prompt and follow it to build and deploy my Amodal agent:
https://docs.amodalai.com/agent-setup/prompt.md
```

The prompt tells the agent to download these docs as text (`llms-full.txt`) and the Platform API reference (`openapi.json`), collect missing configuration, connect source, deploy, and test the result.

Platform Token

The coding agent needs a `pk_` platform automation token to create, configure, or deploy agents:

1. Open Amodal and select your organization.
2. Open **Organization > Automation Tokens**.
3. Create an org-wide `admin` token for initial setup.

Creating an agent requires an `admin` token. Once the agent exists, an agent-bound `operator` token can deploy, activate, promote, roll back, and mint runtime tokens. An `observer` token can only read. Automation tokens cannot create or rotate other automation tokens.

Also give the coding agent the organization ID, the source repository, and any runtime secrets the source references.

A platform token does not authorize chat. Chat smoke tests use a short-lived runtime token minted with `POST /api/agents/{agent_id}/tokens` ; see Runtime Server.

START

Project Structure

An agent source bundle has `amodal.json` at its root. It can contain only agent definitions, or include a frontend app that Amodal builds and serves with the deployed agent. The same layout works in a Git repository and in Amodal-hosted source.

Put agent definition files under the `amodal/` content directory, with hooks, named agents, and evals in their top-level directories:

```
my-agent/
├── amodal.json
├── package.json                # optional, required for custom SPA or custom code deps
├── src/                        # optional custom runtime app
├── index.html                  # optional custom runtime app
├── vite.config.ts              # optional custom runtime app
├── amodal/
│   ├── connections/
│   │   └── s3/
│   │       ├── spec.json
│   │       ├── policy.json
│   │       ├── surface.md
│   │       ├── entities.md
│   │       └── rules.md
│   ├── skills/
│   │   └── ask-agent/
│   │       └── SKILL.md
│   ├── knowledge/
│   │   └── operating-rules.md
│   ├── stores/
│   │   └── investors.json
│   ├── tools/
│   │   └── propose_revision/
│   │       └── tool.ts
│   ├── _lib/                   # optional shared code imported by tools
│   └── _types/                 # optional shared types (not loaded as agent content)
├── hooks/
│   └── redact-pii/
│       ├── hook.json
│       └── index.mjs
├── agents/
│   └── compliance-checker/
│       ├── AGENT.md
│       ├── agent.json
│       └── CONTEXT.md            # optional context rendered each turn
└── evals/
    └── happy-path.md
```

Keep agent content in these reserved directories so it stays separate from app source, scripts, and fixtures. Folders under `amodal/` other than the content types, such as `_lib/` and `_types/`, are not loaded as agent content; use them for code that tools import.

To configure the default chat, add `agents/default/AGENT.md` with the agent's instructions and `agents/default/agent.json` with its capabilities. The prompt file is required; a directory containing only

`agent.json` is ignored.

For example, `agents/default/agent.json` can select a custom tool:

```
{
  "name": "Default",
  "tools": ["propose_revision"]
}
```

The runtime uses `default` when a chat request omits `agent`. Other named agents are selected with `"agent": "<directory-name>"`. Their capability lists determine which skills, connections, stores, and tools are available; see Agents.

Without an authored `default`, chat is unscoped and uses `agents/main/AGENT.md`, when present, as its instructions inside the platform prompt. `agents/main/agent.json` can add opt-in tools to that chat but does not narrow its skills, connections, or stores.

The loader also accepts top-level `connections/`, `skills/`, `knowledge/`, `stores/`, and `tools/`. When `amodal/<type>/` has entries, the top-level directory for that type is ignored. Use one location per type.

Reserved Paths

Path	Purpose
`amodal.json`	Root manifest: identity, packages, memory, web tools, custom runtime app config.
`amodal/connections/<name>/`	API, MCP, or driver-backed connection definitions.
`amodal/skills/<name>/SKILL.md`	Reasoning methods the agent can apply.
`amodal/knowledge/*.md`	Domain reference documents injected or loaded as context.
`amodal/stores/*.json`	Typed store schemas. The runtime generates store tools from these files.
`amodal/tools/<name>/tool.ts`	Custom code tools.
`amodal/_lib/`, `amodal/_types/`	Optional shared code and types imported by tools. Never scanned as agent content.
`hooks/<name>/`	Guardrail hooks: a `hook.json` manifest plus an `index.mjs` handler.
`agents/<name>/AGENT.md`	Instructions for a named agent.
`agents/<name>/agent.json`	The named agent's capabilities and execution settings.
`agents/<name>/CONTEXT.md`	Optional named-agent context rendered each turn.
`evals/*.md`	Evaluation cases.
`src/`, `index.html`, build files	Optional custom runtime app. Used only when `runtimeApp.custom` is enabled.

Outside The Source Bundle

- Installed content packages are declared in `amodal.json#packages`; see amodal.json.
- Scheduled and background automations are configured in Amodal, not in source files.

START

CLI

The `amodal` CLI runs an agent locally, executes evals, and manages hosted deployments. It includes the JavaScript runtime and chat UI. The installed binary does not require Node or npm; agent packages and custom frontend builds may need their own toolchain.

Install

On macOS or Linux:

```
curl -fsSL https://releases.amodalai.com/install.sh | bash
```

The installer uses `curl` and `tar`, installs into `~/.amodal`, and adds the command to your `PATH`. Restart your shell, then run:

```
amodal --version
amodal login
amodal dev ./path/to/agent
```

An agent directory must contain `amodal.json`. Run `amodal <command> --help` for command options.

Running the installer again deletes and recreates `~/.amodal`, which also removes the stored login session and locally stored OAuth credentials. Use `amodal upgrade` to update an existing installation.

Commands

Command	Purpose
`amodal login`	Sign in and select an organization and agent.
`amodal logout`	Remove the stored login session.
`amodal auth token`	Print the access token of the current browser login session.
`amodal org`	Select the active organization; `--list` lists organizations.
`amodal agent`	Select or create the agent used for local trace attribution and deployment commands; `--list` lists agents.
`amodal gateway`	Resolve the cached model gateway key; `--refresh` mints a fresh key.
`amodal dev [path]`	Start the local runtime and chat UI.
`amodal eval [filter]`	Run evals and report failures.
`amodal connections <action> [name]`	Declare, configure, and authorize named connections.

Command	Purpose
`amodal corpus index`	Build semantic indexes for reference corpora.
`amodal deploy`	Build the active agent's connected source.
`amodal deploy-status <deploy-id>`	Read deployment status; `--watch` waits for completion.
`amodal promote`	Promote the active deployment from one environment to another.
`amodal rollback`	Activate an earlier deployment.
`amodal deploy-settings`	Read or update deployment settings.
`amodal upgrade`	Install a published CLI version.

Sign In

```
amodal login
```

Login opens a browser and stores the resulting session in `~/.amodal/credentials.json` with owner-only permissions. It selects an organization and agent, prompting when there are several choices. If the organization has no agent, login automatically creates "Default Agent" for local trace attribution.

Signed-in local runs can use Amodal's model gateway, subject to the selected organization's model access, quota, and billing. Switch the organization or agent with `amodal org` or `amodal agent`. Switching organizations also selects an agent in that organization.

To print the login URL without opening a browser automatically, pass `--no-browser`. To sign in non-interactively, pass a platform token of kind `cli`:

```
amodal login --no-browser
amodal login --token "$AMODAL_CLI_TOKEN"
```

The Automation Tokens page does not offer the `cli` kind. Create one with `POST /api/orgs/{org_id}/automation-tokens` and `"kind": "cli"`. The `observer`, `operator`, and `admin` kinds cannot reach the CLI's organization, agent, gateway, and telemetry endpoints. For deployment automation with an `operator` token, use the Deployment API.

Run Locally

```
amodal dev                          # agent in the current directory
amodal dev ./agent --port 4000      # another directory and port
amodal dev ./agent --open           # also open the chat UI in a browser
```

The runtime API and chat UI share `http://localhost:3847` by default. The local server uses in-memory sessions and stores, so it needs no database and loses local state when it stops.

Choose A Model

Use `--model` or the `MODEL` environment variable to choose the model. The CLI infers `anthropic` for `claude-*`, `google` for `gemini-*`, and an OpenAI-compatible provider for other names. Set `--provider` or `LLM_PROVIDER` to override that inference. Flags take precedence over environment variables.

Without overrides, a gateway run uses the platform's configured model. A run using local credentials defaults to `google/gemini-3.6-flash`. Model availability depends on organization settings and upstream provider access; see Providers.

Gateway Or Provider Keys

With no local key or custom base URL, a signed-in run uses Amodal's gateway. Setting a key for the selected provider uses your credentials instead:

Provider	API key variables
`google`	`GEMINI_API_KEY`, `GOOGLE_API_KEY`, or `GOOGLE_GENERATIVE_AI_API_KEY`
`anthropic`	`ANTHROPIC_API_KEY`
`openai`	`OPENAI_API_KEY`

OpenAI-compatible providers (`groq`, `together`, `fireworks`, `mistral`, `deepseek`) also read `<PROVIDER>_API_KEY`. The CLI stores state under `AMODAL_HOME` (default `~/.amodal`) and reads `AMODAL_PLATFORM_URL` and `AMODAL_DASHBOARD_URL` when targeting a non-default platform environment.

Put variables in your shell or an uncommitted `.env` file. The CLI loads `.env` from the agent directory, then the current directory. Shell variables take precedence; the agent directory takes precedence over the current directory.

An explicit base URL (`--base-url`, `LLM_BASE_URL`, or `OPENAI_BASE_URL`) also selects your own routing. Use `--gateway` to force the platform gateway or `--no-gateway` to use local credentials.

Only the selected provider's key affects routing. An unrelated provider key does not disable the gateway. A gateway provider override that differs from the gateway default also needs an explicit model.

If gateway setup fails, the CLI reports the reason and tries the local configuration. Local model calls still require suitable credentials. Check the startup banner to confirm the model and route.

Hot Reload

The watcher reloads root-level `agents/`, `tools/`, `skills/`, `hooks/`, `connections/`, `knowledge/`, `stores/`, `evals/`, `package.json`, and `amodal.json`. Existing chat sessions keep their history.

Edits inside `amodal/` require a server restart. That directory is loaded as agent content but is outside the local watcher's targets.

Edits to the loaded `.env` files reapply variables and resolve the model again. Invalid bundle edits log a reload error and leave the last working bundle serving. Build output, `node_modules/`, and other dotfiles do not trigger reloads. Use `--no-watch` to disable watching.

Scope

`amodal dev` runs every request under the scope `local`. Choose another scope with:

```
amodal dev ./agent --scope mission-42
```

A `scope_id` in the request body does not override this scope. Agents configured to require a scope therefore run locally without one in the request; see amodal.json.

Traces And Session Sync

Trace export and session sync send local runs to the selected agent's **Traces** and **Sessions** pages, using your login session. Both are on by default for gateway runs and off for runs using your own provider key. Toggle them with `--traces` / `--no-traces` and `--sync-sessions` / `--no-sync-sessions`. The `Traces: exporting to ...` startup line confirms export is active.

Both need a selected agent in the login organization. A one-off `--org` override to another organization disables them; use `amodal org` to switch the stored organization and agent together. Session sync is best effort, and the in-memory store remains the runtime's working store.

Startup Checks

Startup reports the model, routing, scope, and runtime URL. It lists missing provider keys, `env:` references in root-level `connections/*/spec.json`, variables in root-level `tools/*/tool.json` `env` arrays, and uninstalled packages declared in `package.json` dependencies or as plain strings in the `amodal.json` `packages` array. The scan does not read `amodal/connections/` or `amodal/tools/`, so check those definitions yourself when a connection or tool reports a missing variable.

Dev Options

Option	Description
`[path]`	Agent directory; default `.`.
`--port`	Listen port; default `3847`.
`--host`	Bind address; default `127.0.0.1`.
`--model`, `--provider`	Override the model and provider.
`--base-url`	Use a custom model endpoint.
`--gateway`, `--no-gateway`	Force gateway or local routing.
`--org`	Organization for gateway calls in this run.
`--scope`	Scope applied to every request; default `local`.
`--traces`, `--no-traces`	Enable or disable trace export.
`--sync-sessions`, `--no-sync-sessions`	Enable or disable transcript sync.

Option	Description
`--no-watch`	Disable hot reload.
`--oauth-cloud`	Use the selected cloud agent's OAuth bindings, shared with its deployments; default off.
`--open`	Open the chat UI in a browser; default off.
`-V`, `--verbose`	Increase log verbosity.
`-q`, `--quiet`	Show errors only.

Run Evals

```
amodal eval                          # every eval in evals/
amodal eval refund                   # IDs containing "refund"
amodal eval journeys/                # directory prefix, at any depth
amodal eval '/^l[23]\//'              # regular expression
amodal eval --tag safety              # manifest tag; repeat for OR matching
amodal eval --json                    # JSON results
amodal eval --record-dir out/         # training records as JSONL
amodal eval --path ./agent            # another agent directory
amodal eval --url http://localhost:3847
amodal eval --trace-file spans.jsonl # every span as JSONL, for a local run
```

Without `--url`, the CLI starts a local runtime on an ephemeral port and stops it when the run finishes. It accepts the same model-routing flags as `amodal dev`: `--model`, `--provider`, `--base-url`, `--gateway`, `--org`, `--traces`, and `--port`. Evals do not sync their sessions.

`--url` targets an existing server's eval endpoints. The command does not send runtime bearer tokens or browser cookies, so it cannot authenticate to a gated hosted runtime. Use Amodal's Evals page or the Platform API for hosted evals.

The command exits nonzero if any eval fails or no evals run. The suite timeout defaults to five minutes; `--timeout` takes milliseconds. A timed-out suite preserves completed results and reports how many evals finished. Per-eval `timeoutMs` limits still apply.

See Evals for file formats and assertions.

Manage Connections

Each command addresses a connection by its directory name. `add` and `remove` write the working tree at `--path` and do not commit. They use `amodal/connections/` when it contains entries, otherwise `connections/`. The other actions manage OAuth bindings and accounts on the agent selected at login, or in the local store with `--local`. `list` lists those bindings and accounts, not every connection declared in the repository. See OAuth connections for examples and the authorization workflow.

Option	Description
`<action>`	`add`, `remove`, `list`, `configure`, `connect`, `status`, `test`, or `disconnect`.
`[name]`	Connection name; required for every action except `list`.
`--rest`, `--mcp`, `--provider`	`add`: the REST base URL, the HTTP MCP server URL, or the managed OAuth provider ID (`--rest` may accompany `--provider`).
`--auth`	`add`: `bearer`, `api_key`, `basic`, `header`, or `oauth2`; default `bearer` for `--rest`, `oauth2` for `--mcp` and `--provider`.
`--header`	`add`: header name for `--auth header`.
`--secret-name`	`add`: environment variable holding the token; default `<NAME>_API_KEY`, `<NAME>_BASIC_AUTH` for `basic`, `<NAME>_TOKEN` for an MCP server. Names are uppercased, punctuation becomes `_`, and a leading digit gets an `_` prefix.
`--oauth`	`add`: `oauth.json` to copy for a custom OAuth application.
`--agent`	Cloud agent ID; defaults to the agent selected at login.
`--local`	Use the private local store instead of the platform; not with `--managed`. `remove --local` also deletes the local binding.
`--path`	Agent directory written by `add` and `remove` and read by `--local` and `test`; default `.`.
`--config`	JSON application configuration without secrets; required by `configure` unless selecting `--grant` or `--managed`. The CLI does not read the connection's `oauth.json` automatically.
`--grant`	Bind an existing authorized account by ID.
`--managed`	Bind a managed application by ID. A provider with a per-customer host also needs `--rest` with the connection base URL.
`--secret-stdin`	Read the client secret from standard input.
`--callback-port`	Loopback callback port for `--local` `connect`; default `8788`.
`--no-open`	Print the authorization link without opening a browser.
`--json`	Single-line JSON output; the default is indented.

Index Reference Corpora

```
amodal corpus index --path ./agent --app "$AMODAL_AGENT_ID"
```

This embeds `type: corpus` knowledge and stores its search index. `--app` must match the agent ID used by the searching runtime; it defaults to `local`. Configure `CORPUS_EMBEDDING_URL`, `CORPUS_EMBEDDING_API_KEY`, and the chosen vector store first. The PostgreSQL store uses `CORPUS_DATABASE_URL` or `DATABASE_URL`.

Unchanged content reuses cached embeddings. `--json` prints results as JSON. Failed or skipped corpora cause a nonzero exit. See Knowledge Base.

Manage Deployments

Deployment commands build the active agent's connected source. Commit and publish Git changes before deploying; these commands do not upload the local working directory.

```
amodal deploy --wait                                    # production branch into production
amodal deploy --branch feature-x --env preview --wait
amodal deploy-status "$DEPLOY_ID" --watch
amodal promote --from-env preview --to-env production
amodal rollback --env production --deploy-id "$KNOWN_GOOD_DEPLOY_ID"
amodal deploy-settings                                  # print the settings
amodal deploy-settings --production-branch main --auto-deploy --no-auto-promote-production
amodal deploy-settings --root-dir agent                 # folder holding amodal.json; "" for the repo
root
```

`amodal deploy` defaults to the production branch from deploy settings and the `production` environment. Any other environment must be passed explicitly with `--env`. `--message` attaches a deploy message. `--agent <id>` overrides the selected agent for `deploy`, `promote`, `rollback`, and `deploy-settings`. `--wait` and `--watch` stop after ten minutes by default; `--timeout` takes milliseconds.

Promotion, rollback, and deploying a branch other than the production branch into production ask for confirmation. Pass `--yes` in noninteractive scripts. See Deployment API for how building, activating, promoting, and rolling back differ.

Upgrade

```
amodal upgrade                     # latest published version
amodal upgrade --to "$CLI_VERSION"       # a specific published version
amodal upgrade --force              # reinstall the selected version
```

START

Tutorial: Build A Support Agent

Build a support agent that reads example tickets and customers from JSONPlaceholder, follows a triage skill, and saves investigation notes. The demo API needs no connection credentials. Model access is provided by your Amodal organization.

1. Create The Repo

Create a GitHub repo with this tree:

```
amodal.json
amodal/
  connections/support-demo/
    spec.json
    policy.json
    surface.md
  knowledge/
    support-policy.md
  skills/
    triage/SKILL.md
  stores/
    investigation-notes.json
agents/
  default/
    AGENT.md
    agent.json
evals/
  ticket-triage.md
  account-owner.md
```

Connect the repo to an agent in Amodal, or use Build With A Coding Agent to create and connect it with the Platform API.

2. Add `amodal.json`

```
{
  "name": "support-demo",
  "version": "0.1.0",
  "description": "Tutorial agent for support-ticket triage"
}
```

Add the default chat instructions in `agents/default/AGENT.md`:

```
# Support Agent

Help the support operator investigate tickets using the support-demo connection.
Load the Triage skill and the support-policy knowledge document before working.
Check the ticket and its customer before recommending an action. Save an investigation note when the
operator asks.
Do not include customer email addresses in your answer.
```

Select its capabilities in `agents/default/agent.json`:

```
{
  "name": "Default",
  "tools": ["load_skill", "load_knowledge"],
  "skills": ["Triage"],
  "connections": ["support-demo"],
  "stores": {
    "investigation-notes": "rw"
  }
}
```

The runtime selects `default` when a chat request omits `agent`. `AGENT.md` is required; a directory containing only `agent.json` is ignored. Skill entries use the parsed skill name: `Triage` matches the `# Skill: Triage` heading below.

Model selection is platform state, not an `amodal.json` field. If you do not choose a model, the platform default is Gemini 3.5 Flash.

3. Add A Public Connection

`amodal/connections/support-demo/spec.json`:

```
{
  "baseUrl": "https://jsonplaceholder.typicode.com",
  "format": "rest",
  "testPath": "/todos/1"
}
```

This API is public, so there is no `auth` block and no secret to configure.

`amodal/connections/support-demo/surface.md`:

```
# Support Demo API

This is a fake support system backed by JSONPlaceholder.

Use these read-only endpoints:

- `GET /todos/{id}`: load a fake support ticket.
  - `id` is the ticket ID.
  - `userId` is the related customer ID.
  - `title` is the issue summary.
  - `completed` means the issue is resolved.

- `GET /users/{id}`: load the fake customer account.
  - `id` is the customer ID from the ticket's `userId`.
  - Summarize `name` and `company.name`; omit `email`.

When asked about a ticket, first load `/todos/{id}`. If the ticket includes a `userId`, load
`/users/{userId}` before recommending action.
```

`amodal/connections/support-demo/policy.json`:

```
{
  "endpoints": {
    "*": {
      "returns": [],
      "confirm": "never",
      "reason": "This tutorial only reads from the API."
    },
    "GET /todos/{id}": {
      "returns": ["Ticket"]
    },
    "GET /users/{id}": {
      "returns": ["Customer"]
    }
  }
}
```

The fallback `*` rule blocks write requests. The GET entries name expected response entities; they do not limit which GET paths can be requested. The prompt asks the model to omit email addresses. This tutorial does not configure enforced field redaction; see Security before using real customer data.

4. Add Knowledge

`amodal/knowledge/support-policy.md`:

```
---
description: Rules for reporting support investigations and recommending next actions.
---

# Support Policy

- State which ticket and customer records were checked.
- Treat `completed: true` as resolved and `completed: false` as still open.
- Separate confirmed facts from assumptions.
- Do not expose credentials, payment tokens, or private internal notes.
- When data is missing, ask for the missing ticket or customer identifier.
- End with one concise next step for the support operator.
```

The initial prompt lists knowledge documents by name and description. Their bodies are read with `load_knowledge`, which this agent declares in its tools list. Put reference policy here and task instructions in skills. Instructions that must apply before any tool call belong in `AGENT.md`.

5. Add A Skill

`amodal/skills/triage/SKILL.md`:

```
# Skill: Triage

Use this workflow for support-ticket questions.
Trigger: When the user asks about a ticket, customer issue, support case, or next operator action.

## Behavior

1. Identify the ticket ID from the user message.
2. Load `GET /todos/{id}` from the `support-demo` connection.
3. If the ticket has a `userId`, load `GET /users/{userId}`.
4. Summarize confirmed facts:
   - ticket ID
   - issue title
   - open/resolved state
   - customer name or company when available
5. Recommend one next support action.
6. If useful, write a short record to `investigation-notes`.

## Constraints

- Do not invent ticket fields that are not returned by the API.
- Do not call write endpoints; this demo connection is read-only.
- If the API call fails, explain that the ticket could not be loaded.
```

6. Add A Store

`amodal/stores/investigation-notes.json`:

```
{
  "name": "investigation-notes",
  "entity": {
    "name": "InvestigationNote",
    "key": "{ticket_id}",
    "schema": {
      "ticket_id": { "type": "string" },
      "summary": { "type": "string" },
      "next_action": { "type": "string" },
      "confidence": {
        "type": "enum",
        "values": ["low", "medium", "high"]
      }
    }
  },
  "history": { "versions": 3 },
  "trace": true
}
```

The runtime generates store tools from this file, named `store__investigation_notes__<action>`, such as `store__investigation_notes__get` and `store__investigation_notes__set`. Hyphens in the store name become underscores.

7. Add Evals

`evals/ticket-triage.md`:

```
# Eval: Ticket Triage

Tests whether the agent checks ticket data before recommending action.

## Setup

Context: Ticket 1 exists in the demo support API.

## Query

"Triage ticket 1. What should the operator do next?"

## Assertions

- Should load or reference ticket 1 before answering
- Should identify whether the ticket is open or resolved
- Should provide one concise next action
- Should NOT invent private customer data
```

`evals/account-owner.md`:

```
# Eval: Account Owner

Tests whether the agent follows the ticket's userId to the customer record.

## Setup

Context: Ticket 1 has a related userId.

## Query

"Who owns ticket 1 and what is the issue?"

## Assertions

- Should check the ticket record
- Should use the ticket's userId to check the related customer
- Should summarize the issue title
- Should avoid unnecessary personal details
```

8. Deploy

Commit and push the repo. In Amodal:

1. Open the agent.
2. Open **Source** and confirm the source files are visible.
3. Open **Deploys > Source history** and deploy the selected version.
4. Wait for the build and runtime to be ready.
5. Use **Make production** when the tested deploy should serve the persistent agent URL.

Before chatting, check **Connections**. The `support-demo` connection should show the files from `amodal/connections/support-demo/` and the health check should be able to reach `/todos/1`.

9. Smoke Test

Ask these in the deployed chat:

```
Triage ticket 1. What should the support operator do next?

Who owns ticket 1, and what company are they associated with?

Triage ticket 2 and save a short investigation note.
```

The answer should identify the checked ticket, use its returned title and completion state, load the related customer, and give one next support action.

If the answer is generic and does not mention tool data, inspect:

- **Sessions**: confirm there is a `request` tool call.
- **Logs**: check for connection, auth, or provider errors.
- **Prompt**: confirm the `Triage` skill and support policy loaded.
- **Guardrails**: confirm the connection policy blocks writes.
- **Deploys**: confirm production points at the deploy that contains these files.

10. Run Evals

Open **Evals** and run the two eval files. Review the first run before using it as a baseline. After changing source, redeploy and rerun the evals. To check the store, ask the agent to save an investigation note and confirm the `store__investigation_notes__set` call appears in the session.

START

Source & Deploy In Amodal

Use Amodal to create an agent, deploy its source, and inspect the result. For example source to deploy, follow the support-agent tutorial.

1. Choose The Source

Select your organization and create an agent. Choose one of these paths:

- **Template:** select a template and name the agent. Amodal stores its source and opens **Deploys**.
- **Blank agent:** select **Blank agent** and name it. Its source is an `amodal.json` with only a name and a version. Amodal stores it and opens **Deploys**. Tick **Let the agent propose changes to itself** to also add `agents/main/AGENT.md` and an `agents/main/agent.json` that declares `list_repo_files`, `read_repo_file`, and `propose_source_change`. The agent can then propose changes to its own prompt, skills, knowledge, and tool code when asked in chat, and Studio enables the **Builder chat** so you can ask the assistant to scaffold connections, `policy.json` field restrictions, `hooks/` guardrails, knowledge documents, evals, or a custom React runtime app. Each change waits under **Proposals** until someone approves it. See Proposals.
- **Git repository:** choose **Connect a git repository instead**, select the repository, and name the agent. GitHub uses the Amodal GitHub App. GitLab is available when enabled for the platform.

For GitHub, the App installation determines which repositories appear. If yours is missing, use **Configure GitHub App permissions** to grant access.

Template and blank agent source can be exported to an empty GitHub repository from **Settings > Source**. You can also keep editing it in Amodal.

2. Configure The Agent

Before deploying, check:

- **Models:** choose the model. The hosted default is Gemini 3.5 Flash.
- **Organization settings:** configure provider credentials and allowed models when needed.
- **Secrets:** set connection credentials and other values referenced with `env:NAME`.
- **Auth, embed, and domains:** configure app access and embedding.
- **Runtime and deploy settings:** review runtime resources, source settings, and automatic deployment behavior.

The model choice and secret values are platform settings. Keep secret references in source; keep the values in Amodal.

3. Deploy And Select Production

Open **Deploys**, select the source version, and deploy it. Wait for the build to finish, then check the deploy's runtime status and URL.

Source versions and deployments have separate histories. One source version can have several deployments. With manual production promotion, the default setting, a completed production build stays inactive until you promote it. Turn on **Auto-promote to production** in deploy settings if you want completed production builds to go live automatically. Automatic deployment on push can also start a build without making it production.

Open the deploy URL to test that version before promoting it, then use **Make live** (**Make production**) and check the production indicator. The persistent agent URL serves the active production deployment. See Deployment API for API activation, promotion, and rollback.

4. Chat And Inspect

Ask a question that exercises one of the agent's connections or tools. In the support tutorial, use:

```
Triage ticket 1. What should the support operator do next?
```

Inspect the messages and tool calls in **Sessions**, and the model, token usage, and timing in **Traces**. Check **Logs** for runtime errors. Before the first deployment, runtime pages may have no data.

5. Edit, Test, And Redeploy

Edit the source files and review the diff before committing. For a connected Git repository, publish the changes to the branch you intend to deploy. For Amodal-hosted source, save a source version in Amodal.

Deploy the changed source, run the agent's evals, and test the deploy URL. Select production after checking the result.

AGENT REPO

amodal.json

Every Amodal agent repo has an `amodal.json` file at the repo root. It names the agent, declares package content, and enables optional runtime features.

```
{
  "name": "investor-relations",
  "version": "0.1.0"
}
```

Model selection and provider credentials are platform settings, not `amodal.json` fields. Select a model in Amodal or through the Platform API.

The schema drops unknown fields without an error, so a misspelled field has no effect.

Fields

Field	Type	Description
`name`	string	Required. Agent name used in logs, deploy metadata, and generated identifiers.
`version`	string	Required. Non-empty agent source version that you control.
`schemaVersion`	integer	Manifest format version. Defaults to `1`. A value newer than the runtime supports fails with `CONFIG_SCHEMA_VERSION_UNSUPPORTED`.
`description`	string	Human-readable description.
`basePrompt`	string	Replaces the compiled base system prompt. Prefer skills and knowledge for behavior and domain context.
`packages`	array	npm packages that ship agent content. See Packages.
`runtimeApp`	object	Custom browser app build settings: `custom`, `build`, `dist`. See Runtime Apps.
`mcp.servers`	object	MCP server definitions. See MCP Servers.
`webTools`	object or `true`	Enables `web_search` and `fetch_url` through Google grounding. See Tools.
`memory`	object	Persistent memory across sessions. See Memory.
`scope`	object	`requireScope` flag. See Scope.
`fileTools`	boolean or object	File-tool sandbox settings: `allowedDirs` and `blockedFiles`. `false` disables file tools. Agents still opt in through their `tools` list.
`hooks`	object	Hook `order`, `disabled`, and per-hook `config`. See Hooks.

Field	Type	Description
`ci`	object	Push deployment rules (`ci.deploy`) and deployment evals (`ci.evals`). See CI Rules.
`sandbox`	object	Sandbox for agent code. `provider` (for example `local`) gives the agent the `bash` , `read_file` , `write_file` , and `publish_file` tools, each session in its own sandbox. Also accepts `setup` (shell script run when the environment is built), `network` (`deny-all` by default, `allow-all` , or `{ "allow": [...] }`), and adapter `config` . `shellExec` , `template` , and `maxTimeout` have no runtime effect.
`artifactProviders`	array	App-owned HTTP artifact sources (`{ id, baseUrlEnv, tokenEnv, path }`) that custom tools can register via `ctx.artifacts.register` .

Store schemas live in `amodal/stores/*.json` . Do not put store backend configuration in `amodal.json` ; the cloud runtime supplies the database.

Packages

```
{
  "packages": [
    "@acme/connection-crm",
    {
      "package": "@acme/support-content",
      "use": ["skills.triage", "knowledge.support-policy"]
    }
  ]
}
```

A bare string loads everything the package ships. The object form loads only the `use` entries, each written as `<kind>.<name>` . Kinds are `connections` , `skills` , `knowledge` , `stores` , `tools` , `hooks` , and `channels` . Local repo content wins when names collide.

Install the packages with your package manager. npm dependencies for a runtime app or tool code belong in `package.json` , not in `amodal.json` .

Environment Variables

Write a string value as `env:NAME` to read it from the environment instead of committing it:

```
{
  "webTools": {
    "provider": "google",
    "apiKey": "env:GOOGLE_GENERATIVE_AI_API_KEY"
  }
}
```

`amodal dev` resolves every `env:` value in `amodal.json` when it loads the file. An unset variable fails the load with `ENV_NOT_SET`.

The hosted runtime loads `amodal.json` without resolving `env:` values. Connection specs and `webTools.apiKey` resolve their own `env:` references from agent secrets; see Connections.

`"webTools": true` is shorthand for the example above.

Memory

```
{
  "memory": {
    "enabled": true,
    "editableBy": "any",
    "maxEntries": 50,
    "maxTotalChars": 8000
  }
}
```

With `enabled: true` and a memory backend, the runtime registers a `memory` tool with `add`, `remove`, `list`, and `search` actions. A named agent must also list `memory` in its `tools`. `editableBy` defaults to `"any"`; `"admin"` and `"none"` both disable the tool, and saved memory still appears in the prompt. `maxEntries` defaults to `50` and `maxTotalChars` to `8000`.

Memory is stored per agent and partitioned by `scope_id`. The schema also accepts `nudgeInterval` and `sessionSearch`, which have no runtime effect.

Scope

`scope_id` partitions sessions, memory, and non-shared store data by end user, tenant, or workspace. Your backend sets it in the runtime tokens it mints; see Embedding & Multi-tenancy.

The hosted runtime does not enforce `scope.requireScope`. `amodal dev` runs every request in scope `local`, or in the value of `--scope`, and ignores a `scope_id` in the request body. Enforce scope in your backend instead of relying on this flag.

CI Rules

The platform reads `ci` from `amodal.json` at the pushed commit. `ci.deploy` applies to connected GitHub and GitLab repositories. `ci.evals` applies to GitHub only.

```
{
  "ci": {
    "deploy": {
      "branches": ["main", "release/*"],
      "paths": ["package.json", "pnpm-lock.yaml"]
    },
    "evals": { "runOn": "production" }
  }
}
```

Field	Behavior
`ci.deploy.branches`	Branch names allowed to deploy on push. `*` matches any characters. Omitted means every branch.
`ci.deploy.paths`	Repo-relative path prefixes that trigger a deploy in addition to the agent's root directory.
`ci.evals`	Deployment eval settings: `runOn`, `runs`, `passRate`, `judgeModel`. See Deployment triggers.

A push that touches neither the agent's root directory nor a listed path skips the build. An unreadable `amodal.json` falls back to the defaults.

Pushes build only when `autoDeploy` is on in the agent's deploy settings. It is off by default. See Deploy Settings.

AGENT REPO

Agents

Each directory under `agents/` at the repo root (a sibling of `amodal/`) defines one agent. A named agent can root a conversation or run as a delegated specialist. The directory name is the agent's identifier.

```
agents/
├── main/
│   └── AGENT.md
└── compliance-checker/
    ├── AGENT.md        # prompt, required
    ├── agent.json      # configuration, optional
    ├── agent.ts        # configuration with predicates, optional
    └── CONTEXT.md      # per-turn prompt, optional
```

A directory without `AGENT.md` is ignored.

Reserved Names

Name	Purpose
`default`	Implicit conversation root when a chat request omits `agent`. Its resource declarations apply.
`main`	Operating instructions for unscoped chat when no authored `default` exists. Supports `agents/main/AGENT.md` or `agents/main.md`.
`simple`	Loaded from `agents/simple/AGENT.md` or `agents/simple.md`, but no runtime path reads it. Use a named agent with `modelTier`.

Unscoped chat inserts `main` into the compiled system prompt alongside platform instructions and resource descriptions. A `basePrompt` in amodal.json replaces the compiled prompt instead.

`agents/main/agent.json` can opt unscoped chat into file tools and `propose_source_change` through `tools`. Its other fields are ignored, so `main` does not restrict resources. The file is read only when `agents/main/AGENT.md` exists and is not empty.

Rooting A Conversation

Pass the directory name in the chat request's `agent` field:

```
{
  "message": "Review transactions TXN-001 through TXN-050.",
  "agent": "compliance-checker"
}
```

The root agent's prompt replaces the `main` operating instructions. Its `tools`, `skills`, `connections`, `stores`, `mcp`, and `subagents` declarations select the available resources. Files under `amodal/` still load as repo content, but the session only exposes the names the agent lists. The runtime still supplies chat protocol tools such as `stop_execution` and `ask_choice`.

- An unknown `agent` name fails the request. It does not fall back to the default chat.
- When `agent` is omitted, chat uses `agents/default/` if it exists. Without it, chat is unscoped and exposes every bundle resource.
- An end user's autonomous run of `main` uses `agents/default/` the same way chat does. Operator callers keep the unscoped surface.
- A later turn on the same session that names a different `agent` re-roots the session. The transcript is kept, and the runtime inserts an `[Agent handoff] <from> -> <to>` user message that lists the tools the new agent lacks.
- The active root and `default` are excluded from the callable specialist list.

Skill entries use the loaded skill name, not the folder name. The `# Skill:` heading wins, then frontmatter `name`. An `amodal/skills/craft-greeting/SKILL.md` that starts with `# Skill: Craft Greeting` is referenced as `"Craft Greeting"`. Store entries produce tools named `store__<name>__<op>`, with hyphens in the store name replaced by underscores.

agent.json

```
{
  "name": "Compliance Checker",
  "description": "Check transactions against the supplied compliance policy.",
  "tools": ["load_knowledge", "store__research_notes__query"],
  "stores": { "research-notes": "read" },
  "maxToolCalls": 15,
  "modelTier": "advanced"
}
```

`AGENT.md` holds the prompt. If it has a `## Prompt` section, that section is the prompt; otherwise the whole file is.

Field	Default	Meaning
`name`	Directory name	Display name. The directory name remains the invocation identifier.
`description`	`name`	Description shown in the specialist list.
`tools`	`[]`	Explicit tool names. An agent gets only the tools it names.
`skills`	None	Skills included when the agent roots a session. Each adds its `allowedTools`.
`connections`	None	REST connections for a root session. Native OpenAPI specialists also prepare their own declared connections.

Field	Default	Meaning
`mcp`	None	MCP servers exposed when the agent roots a session.
`subagents`	None	Specialists callable from a session rooted in this agent.
`stores`	None	Per-store `"read"` or `"rw"` access for a root session. `read` grants `get`, `query`, and `list`; `rw` adds `set` and `remove`.
`maxToolCalls`	`10`	Turn limit for a delegated run. One turn may call several tools. Positive integer.
`modelTier`	Session model	`simple`, `default`, or `advanced`. See Model Tiers.

`maxDepth`, `timeout`, `targetOutputMin`, and `targetOutputMax` are validated but have no runtime effect. State the required output format in the prompt.

List entries accept a name or `{"name": "..."}`. Store entries accept `"read"`, `"rw"`, or `{"mode": "read"}`. A `conditional` key in `agent.json` is a load error.

Configuration In AGENT.md

Without `agent.json` or `agent.ts`, `AGENT.md` can carry configuration in YAML frontmatter. Use `displayName` for the display name. The prompt is the body after the frontmatter and must not be empty.

```
---
displayName: Vendor Lookup
description: Look up vendor records and summarize contract status.
tools: [request]
maxToolCalls: 10
modelTier: simple
---

Query the vendor system for the named vendor. Return the company name,
contract status, source record, and any missing information.
```

A `## Config` YAML section is not parsed.

agent.ts

A sibling `agent.ts` accepts the same fields as `agent.json` and adds synchronous predicates to resource entries. It needs no Amodal package dependency:

```
const isManager = (ctx: { claims: Record<string, string> }) => ctx.claims['role'] === 'manager';
const isJoint = (ctx: { context: Record<string, string> }) => ctx.context['mission_type'] === 'joint';

export default {
  tools: ['review', 'check_conflicts'],
  skills: ['request-review', { name: 'joint-review', conditional: isJoint }],
  subagents: ['prescreen', { name: 'adjudicator', conditional: isManager }],
  stores: {
    requests: 'read',
    decisions: { mode: 'rw', conditional: isManager },
  },
};
```

List entries accept a name or `{name, conditional}`. Store entries accept `"read"`, `"rw"`, or `{mode, conditional}`. Any other key in an entry is a load error.

A predicate decides whether a declared entry is included for this caller. It cannot add an undeclared resource. It must return `true` to include the entry. A thrown error or missing caller context excludes it.

```
interface AgentSurfaceContext {
  claims: Record<string, string>;
  context: Record<string, string>;
  scopeId: string;
  userId?: string;
  orgId?: string;
  humanPresent: boolean;
  isSubagent: boolean;
  agentName: string;
}
```

Field	Meaning
`claims`	Verified JWT claims. Use for authorization decisions.
`context`	Merged request context, including client-supplied values. Use to select relevant workflows or resources.
`scopeId`	The session's scope identifier. Empty for agent-level state.
`userId`, `orgId`	Verified caller and organization identifiers, when available.
`humanPresent`	`false` for cron, webhook, and channel runs where nobody can answer a confirmation.
`isSubagent`	`true` when the agent runs as a delegated specialist.
`agentName`	Name of the session's root agent, including in specialist predicates.

A client can send `{"role": "manager"}` in request context. A permission predicate must therefore read `ctx.claims.role`, not `ctx.context.role`.

A delegated specialist inherits the session's surface context with `isSubagent` set to `true`. A curation predicate that reads page state should allow delegation explicitly:

```
{name: 'lookup', conditional: (ctx) => ctx.isSubagent || ctx.context.page === 'requests'}
```

Rules:

- Predicates must be synchronous and pure. Do not fetch data or change state in them.
- Excluding a skill also excludes the tools contributed by its `allowedTools`, unless another declaration grants them.
- `agent.ts` fields override matching `agent.json` fields and log `agent_definition_overrides_json`. Keep each field in one file.
- `agent.ts` requires a default export. Named exports are a load error.
- `AGENT.md` is still required.
- Conditional entries resolve at session creation and are checked again on each turn. If the resolved set changes, the runtime rebuilds the session's resources and keeps the conversation.

CONTEXT.md

`AGENT.md` renders once when the session is built, so values in it stay fixed. Put facts that change while the user works in `CONTEXT.md`, which renders on each turn after the cached prompt prefix.

Both files use Nunjucks and read request context as `scope`:

```
{% if scope.request_status %}
Request status: {{ scope.request_status }}.
{% endif %}
```

Send the context with each chat request:

```
{
  "message": "Why has this not been approved?",
  "agent": "requests",
  "context": {
    "mission_id": "m-42",
    "request_status": "Pending review"
  }
}
```

- `CONTEXT.md` is read when its agent roots the conversation, including `default`. It is not read for `main` or for a delegated specialist.

- Keys must start with a letter or underscore and contain at most 64 letters, digits, or underscores. Other keys are dropped.
- Values are strings and inert text. Template syntax inside a value is not executed. `"false"` and `"0"` are truthy.
- Empty and whitespace-only values are dropped.
- Values have no length cap. The runtime logs `prompt_template_scope_large` above 100,000 total characters.
- Wrap literal template syntax in `{% raw %}...{% endraw %}`.

Prompt text does not enforce permissions. Use a predicate on verified claims to gate a capability, and a tool call for data that must be checked against a server.

Delegation

The root agent calls `call_subagent` with one or more named tasks:

```
call_subagent({
  calls: [
    {
      subagent: 'compliance-checker',
      task: 'Check transactions TXN-001 through TXN-050 against the compliance policy.',
      input: { policy: 'aml-2026' },
    },
  ],
});
```

- Each call accepts at most 12 tasks and runs at most five concurrently.
- A single task returns `{subagent, result}` or `{subagent, error}`. A batch returns `{results: [...]}` in call order. One failed specialist does not fail the batch.
- Nested delegation stops at five levels.
- Specialists run without an interactive approval handler. A tool call that requires confirmation is denied.
- A failing tool returns an error to the specialist's model. A provider failure fails the specialist.
- From 65% of `maxToolCalls`, each tool result carries a `_budget` notice with the turns left. Non-object results are wrapped as `{result, _budget}`.

Context Isolation

A specialist receives its own prompt as the system prompt, the task, the optional `input` appended as a JSON block, and any attachments a composite tool forwards. It does not receive the parent's prompt, skill bodies, or conversation history. Name every tool it needs, including `load_knowledge` or `load_skill`. A declared tool is copied from the parent session's registry, so it must exist there. Declared file tools are the exception: the runtime builds them for the specialist. See Tools for tool names.

Model Tiers

The deployment binds `simple` and `advanced` to models through Studio model settings or the `MODEL_SIMPLE` and `MODEL_ADVANCED` environment variables. `default` and an unbound tier use the session model. `ctx.callSubagent` from a durable tool ignores `modelTier` and uses the session model.

AGENT REPO

Connections

A connection tells the runtime how to call an external API and tells the model which calls exist. Define local connections in `amodal/connections/<name>/` or the top-level `connections/<name>/`. The loader reads `amodal/connections/` when it contains entries, otherwise `connections/`. `amodal connections add <name>` follows the same rule; see Use the CLI.

Where changes are saved

Entry point	Destination
Amodal with a GitHub source	A commit on the selected branch. A branch that does not exist is created from the production branch.
Amodal with an Amodal source	A source snapshot stored by Amodal.
CLI `connections add` or `remove`	Files in the working tree at `--path`. Commit and push them yourself.

Source holds only connection configuration. Secret values, OAuth client secrets, tokens, and account bindings stay in credential storage. Deploy the saved source to make a connection available to a hosted runtime.

Directory Structure

```
amodal/connections/my-api/
├── spec.json       ← protocol, base URL, auth, OpenAPI source
├── policy.json     ← endpoint permissions, confirmations, field restrictions
├── oauth.json      ← (custom OAuth) application settings, never the client secret
├── openapi.json    ← API contract when openapi.source selects this file
├── surface.md      ← endpoint documentation for the request tool
├── entities.md     ← (optional) entity definitions
└── rules.md        ← (optional) business rules
```

A REST connection is served either by native OpenAPI operation tools or by the generic `request` tool. `surface.md` documents endpoints for `request`; native connections derive their tools from the OpenAPI document and ignore it. `policy.json` is required for a local REST connection, and a REST connection without one is skipped with a warning. The minimal policy is `{"endpoints": {}}`. MCP connections need no policy.

Credentials

Credential values live outside the repository. `auth.token`, `auth.privateKey`, and any `headers` or `env` entry named like a credential (`Authorization`, `Proxy-Authorization`, `X-Api-Key`, `Cookie`, or a name ending in `token`, `key`, `secret`, or `password`) must hold `env:NAME`, `scope:KEY`, or a `{{NAME}}` template. A literal value fails validation with `CONNECTION_SECRET_IN_SOURCE`. Other headers and environment entries accept literals.

OAuth 2.0

Declare `auth: {"type": "oauth2"}` in a REST or HTTP MCP connection to authorize an account and refresh its tokens automatically. A custom application is described in `oauth.json`, and a managed provider is selected with `auth.provider`. An MCP server that publishes OAuth metadata needs neither. See OAuth connections.

spec.json

Built-in protocols are `rest` (the default) and `mcp`. Installed connection drivers can define other protocols.

REST Connection

A native OpenAPI connection:

```
{
  "baseUrl": "env:SHELTER_API_URL",
  "openapi": {
    "source": "./openapi.json",
    "exposure": "auto",
    "sessionHeader": "x-agent-session-id"
  },
  "auth": { "type": "bearer", "token": "env:SHELTER_API_TOKEN" },
  "filter": { "exclude_paths": ["/health"] }
}
```

The `openapi` block activates native tools. The runtime prepares the document when it builds a session, registers one tool per selected operation, and withholds the connection from `request`. A missing or invalid document fails preparation; the runtime does not fall back to `request`. A document file, `format`, or `specUrl` without the `openapi` block does not activate native tools.

A minimal `request` connection is a `baseUrl` plus an empty policy:

```
// spec.json
{ "baseUrl": "https://api.adviceslip.com" }
```

```
// policy.json
{ "endpoints": {} }
```

A named agent sees a connection only when its `agent.json` `connections` list includes it. Omit `auth` for a public API; there is no `"none"` auth type.

REST calls through native tools and `request` refuse redirects, cap responses at 5 MiB, and time out after 30 seconds. Set `baseUrl` to the final API URL.

MCP Connection

```
{
  "protocol": "mcp",
  "transport": "stdio",
  "command": "uvx",
  "args": ["mcp-server-github"],
  "env": { "GITHUB_TOKEN": "env:GITHUB_TOKEN" }
}
```

See MCP Servers for transports and options.

Fields

Field	Description
`protocol`	`"rest"` (default) or `"mcp"`.
`baseUrl`	API base URL. Required for REST.
`specUrl`	Documentation URL. Does not activate native tools.
`skipRequestTool`	Withhold a REST connection from `request`, for example when custom tools own API execution. Does not disable native tools.
`test`	Request for **Test connection**: `path` relative to `baseUrl`, `method` (`GET` default, `POST`, or `HEAD`), `query`, `body`, `headers`, and `expectStatuses` (default `[200]`).
`testPath`	Path for a `HEAD` test request when `test` is absent, for example `"/me"`.
`format`	`"openapi"`, `"graphql"`, `"grpc"`, `"rest"`, or `"aws-api"`. REST only.
`auth.type`	`"bearer"`, `"api_key"`, `"basic"`, `"header"`, `"body"`, `"google_service_account"`, or `"oauth2"`. MCP supports `bearer`, and `oauth2` over HTTP or SSE. `oauth2` takes no `token`.
`auth.token`	Credential reference for `bearer`, `api_key`, `basic` (base64 `user:pass`), and `header`.
`auth.header`	REST only. Header for `bearer`, `api_key`, and `header`. Defaults to `Authorization`, or `X-API-Key` for `api_key`.
`auth.prefix`	REST `bearer` prefix (default `Bearer`). MCP `bearer` always sends `Authorization: Bearer <token>`.
`auth.fields`	For `body`: credential fields merged into the JSON body of POST, PUT, and PATCH requests from the `request` tool, and of every non-GET, non-HEAD request from native operation tools.
`auth.clientEmail`	For `google_service_account`, with `auth.privateKey`, `auth.scope` (default `https://www.googleapis.com/auth/cloud-platform`), and `auth.tokenUri` (default `https://oauth2.googleapis.com/token`).

Field	Description
`auth.provider`	Managed OAuth application ID (`oauth2` only).
`filter`	Native operation selection: `tags` and `include_paths` intersect; `exclude_paths` applies last.
`transport`	`"stdio"` , `"sse"` , or `"http"` . MCP only.
`command` , `args`	Command and arguments to spawn. MCP stdio only.
`env`	Environment variables. MCP stdio only.
`url`	Server URL. MCP SSE and HTTP only.
`headers`	HTTP headers. REST, and MCP SSE and HTTP.
`contextInjection`	Forward request context values into API requests. See Context Injection.
`openapi.source`	`./file.json` , `./file.yaml` , `./file.yml` , an HTTP(S) URL, or `env:VAR` resolving to a URL. Files must stay inside the connection directory after symlink resolution.
`openapi.exposure`	`auto` (default), `direct` , or `discovery` . See Discovery.
`openapi.sessionHeader`	Header that carries the runtime session ID on operation requests. Credential header names are reserved. Document fetches do not receive it. Operations of this connection run one at a time.

Context Injection

`contextInjection` forwards values from the chat request's `context` into calls to this connection, such as a tenant or workspace ID:

```
{
  "baseUrl": "https://api.example.com",
  "auth": { "type": "bearer", "token": "env:API_TOKEN" },
  "contextInjection": {
    "tenant_id": { "in": "header", "field": "X-Tenant-Id", "required": true },
    "org_id": { "in": "query", "field": "org_id" }
  }
}
```

Each key names a key in the request context:

```
{
  "message": "What are my recent orders?",
  "scope_id": "user-123",
  "context": { "tenant_id": "tenant-abc", "org_id": "org-456" }
}
```

Field	Description
`in`	`"header"`, `"query"`, `"path"`, or `"body"`.
`field`	Header name, query parameter, `{placeholder}` in the endpoint path, or body field.
`required`	When `true`, a request fails if the context lacks the key. Default `false`, which skips the key.

- `query` is not appended when the request already sets that parameter.
- `body` adds the field to the JSON body, which is sent only for POST, PUT, and PATCH. The `request` tool rejects a call whose `data` is not an object when body injection is configured.
- Native OpenAPI connections support header, query, and body injection. Path injection fails preparation. Native query injection supplies defaults that model arguments can override. Model values for injected body fields are rejected.

Context can contain client-supplied values and is not authenticated identity. The downstream API must validate tenant access. See Scope.

Native OpenAPI connections

Tools and arguments

Operation tools are named `<connection>__<operationId>`. Characters other than letters, digits, underscores, and hyphens become underscores. Names longer than 64 characters are shortened with a hash suffix. A missing `operationId` is derived from the method and path. Duplicate identifiers or tool names fail preparation.

Arguments contain only the declared parameter groups:

```
{
  "path": { "residentId": "r-2" },
  "query": { "notify": false },
  "body": { "adopterName": "Dana Kim" }
}
```

The model cannot select a method, URL, connection, header, or operation through arguments. Arguments that violate the schema fail with `Invalid parameters:` before hooks, policy, confirmation, and HTTP.

Supported contracts:

- OpenAPI 3.0.x and 3.1.x, with GET, HEAD, POST, PUT, PATCH, and DELETE.
- Path parameters with primitive values and `simple` serialization.
- Query parameters with primitives or primitive arrays and `form` serialization.
- JSON bodies, including `application/*+json` media types and DELETE bodies.
- Internal JSON Pointer references. External or cyclic references fail.
- OpenAPI 3.1 tuples with `prefixItems` and `items: false`. Draft-style `items: [...]` tuples are rejected.
- Schemas without a `__proto__` property.

Limits are 16 MiB per document, 2,000 operations, reference depth 64, and 1,024 characters per regex pattern. An unsupported selected operation fails preparation; exclude it with `filter`.

Document `servers`, `security`, and vendor extensions do not route requests, supply credentials, or grant permissions. Configure credentials in `spec.json` and permissions in `policy.json`.

Discovery

Every native connection exposes `<connection>__discover`:

- Without arguments, it returns the tag index.
- `query` searches operation IDs, tool names, paths, tags, and summaries, up to 256 characters.
- `tag` selects a tag. Pages hold at most ten operations; `cursor` fetches the next page.
- Returned operations become typed tools on the next model reply. Calling them in the same reply is refused.
- `refresh: true` reloads the source. It is refused while native calls are pending.

At most 40 native operation tools are visible per model round across native connections. `direct` connections that exceed the budget fail session preparation. `auto` connections use direct mode when they fit beside the direct connections, and discovery otherwise. Discovery evicts the least recently used operations.

Model provider compatibility

Operation schemas are sent to the model as formal tool definitions. The runtime validates arguments against the original schema regardless of how the provider adapter translates it. See Providers.

Policy and confirmation

Native operations differ from `request` in these ways:

- A write without a matching policy entry is denied.
- GET and HEAD are reads. A matching entry with `"intent": "read"` makes another method a read.
- All matching keys apply, including wildcards, and the strictest tier wins. Keys match the template and the encoded concrete path.
- `review` and `never` also deny reads.
- `body` in an entry lists allowed top-level body keys. Lists from several matching entries intersect, and an empty list permits no body. Other keys are rejected before confirmation.

```
{
  "endpoints": {
    "POST /residents/*": { "returns": [], "confirm": true },
    "POST /residents/{residentId}/adopt": { "returns": [], "body": ["adopterName"] }
  }
}
```

Approval is bound to the call and its effective arguments. A hook rewrite after approval asks again.

Freshness and results

File sources are read on every session preparation. URL sources are cached for 60 seconds per URL and credential set, and failures are not cached. A session keeps its catalog until `refresh`. Document fetches time out after 15 seconds and receive connection credentials only on the `baseUrl` origin.

Results over 16 KiB are kept in session memory and returned as a digest with a handle. `read_spilled_result` reads a nested dot path and pages arrays with `offset` and `limit`. The store keeps at most 16 results or 32 MiB and does not survive a process restart.

Custom tools

Custom handlers can call any connection with `ctx.request()`. These calls apply connection authentication, context injection and field restrictions, but not endpoint policy, schema validation, or confirmation. See Tools.

policy.json

```
{
  "endpoints": {
    "GET /tickets/{id}": {
      "returns": ["Ticket"]
    },
    "POST /tickets/{id}/comments": {
      "returns": ["Ticket"],
      "confirm": true,
      "reason": "Adds a visible support-ticket comment"
    },
    "DELETE /tickets/{id}": {
      "returns": ["Ticket"],
      "confirm": "never",
      "reason": "Ticket deletion is not allowed through the agent"
    }
  },
  "fieldRestrictions": [
    {
      "entity": "Customer",
      "field": "payment_token",
      "policy": "never_retrieve",
      "sensitivity": "secret",
      "reason": "Payment tokens are never exposed to the agent"
    },
    {
      "entity": "Customer",
      "field": "email",
      "policy": "retrieve_but_redact",
      "sensitivity": "pii_name"
    }
  ],
  "alternativeLookups": [
    {
      "restrictedField": "Customer.payment_token",
      "alternativeEndpoint": "GET /customers/{id}/billing-status",
      "description": "Use billing status instead of raw payment credentials"
    }
  ],
  "delegations": { "escalateConfirm": true }
}
```

Field	Description
`endpoints`	Required. Keys are `METHOD /path`, with `{param}` and `*` patterns.
`endpoints.*.returns`	Required. Entity names the response contains, used by field restrictions. Use `[]` when none apply.
`endpoints.*.confirm`	Omitted, `true`, `"review"`, or `"never"`. See Action Tiers. `false` is invalid.
`endpoints.*.reason`	Text shown in a confirmation card or denial.
`endpoints.*.thresholds`	Parameter rules that raise the tier. See Threshold Escalation.
`endpoints.*.intent`	`"read"` lets a mutating method run as a read.
`endpoints.*.body`	Allowed top-level body keys. Native OpenAPI only.
`fieldRestrictions`	`entity`, `field`, `policy`, `sensitivity`, and optional `reason`. See Field Restriction Policies.
`alternativeLookups`	Prompt guidance of the form "Instead of `restrictedField`, use `alternativeEndpoint`".
`delegations.escalateConfirm`	When `true`, `confirm` becomes `review` for calls made in a delegated session (a subagent or an autonomous run).

Action Tiers

`confirm`	Behavior
omitted	Run without approval.
`true`	Pause for user approval, then run.
`"review"`	Refuse with a reason stating human review is required. The runtime does not queue the call.
`"never"`	Refuse.

For the generic `request` tool:

- Tiers apply only to writes (POST, PUT, PATCH, DELETE). Reads run without a policy check.
- A write that matches no entry runs without approval. Add a `"*"` or wildcard entry to set a minimum tier.
- An exact key wins. Otherwise the strictest matching pattern applies.

Confirmation Model describes the approval flow and how custom-tool tiers map to these values.

Field Restriction Policies

Policy	Effect
`never_retrieve`	Field removed from the JSON response.
`retrieve_but_redact`	Value masked: `****6789` for a value of 8 or more characters ending in 4 digits, `****` otherwise.

Policy	Effect
`role_gated`	Field always removed. `allowedRoles` is accepted but not applied.

Restrictions also add "Do not request" and "Will be redacted" lines to the system prompt.

For `request` and `ctx.request()` calls, the scrubber checks the top-level object, or each item of a top-level array, against every entity in the endpoint's `returns`. A key named after an entity, such as `customer` or `tickets`, limits everything under it to that entity's restrictions. Other keys, such as `data` or `items`, keep the combined set. Entity names match regardless of case.

Native OpenAPI connections identify entities from the response schema: data at a position whose schema references `#/components/schemas/Taxpayer` is a `Taxpayer`, whatever its key. Restrictions apply to every `oneOf`, `anyOf`, and `allOf` branch, even without a `returns` entry. Key-name matching applies where the schema names no restricted component. On native OpenAPI connections, `ctx.request()` identifies entities from `returns` and key names, not from the response schema.

`ctx.request()` responses are scrubbed with the same rules before the handler receives them, so a custom tool cannot read a restricted field through `ctx.request()`. MCP and connection-driver tool results are not scrubbed: restrictions bind to an endpoint's `returns`, and those results have no endpoint. Guard them with a `postToolUse` hook. Verify the response shape in an eval before relying on removal or redaction.

Threshold Escalation

```
{ "field": "body.amount", "above": 10000, "escalate": "review" }
```

Add this object to an endpoint's `thresholds` array. When `body.amount` exceeds 10000, the tier escalates to `review` and the call is denied. `escalate` accepts `"review"` or `"never"`. A missing or non-numeric value escalates as if the threshold were exceeded, and numeric strings compare as numbers. `field` is a dot path into the grouped parameters: `body` and `query`, plus `path` for native tools. Native thresholds see values after context injection.

Packaged Connections

Packages listed in `amodal.json#packages` can ship connections. See amodal.json. A local connection with its own `spec.json` takes precedence over a packaged one of the same name. To customize a packaged connection while keeping its `spec.json`, add `surface.md`, `policy.json`, `oauth.json`, `entities.md`, or `rules.md` under the top-level `connections/<name>/`, not under `amodal/`. A local `policy.json` merges with the package policy.

AGENT REPO

OAuth connections

OAuth 2.0 is available for REST connections and for MCP connections over `http` or `sse`. Stdio MCP servers and external connection drivers do not support it. The runtime acquires and refreshes bearer tokens during requests. Tokens and client secrets stay in encrypted credential storage and never become environment variables.

An authorized account belongs to the organization. Each agent connection has one account binding, shared by its deployments and by cloud-backed local development. Other connections use that account only when an administrator binds them to it.

Declare OAuth in the connection

Set `auth` in `spec.json`:

```
{
  "baseUrl": "https://gitlab.com/api/v4",
  "auth": { "type": "oauth2" },
  "test": { "path": "user" }
}
```

`auth.token` is rejected with `oauth2`. The application configuration comes from the first of:

1. `auth.provider`, the id of a managed provider. The CLI cannot use managed providers with `--local`.
2. `oauth.json` next to `spec.json`, for a custom application.
3. Discovery from the server URL, for an MCP connection whose server publishes OAuth resource metadata (RFC 9728) and registers public clients (RFC 7591).

A REST OAuth connection with none of these is listed as needing configuration. An existing approved binding takes precedence over the authentication declared in the files, so editing `spec.json` alone does not remove it.

`oauth.json`

`oauth.json` holds the application settings. It never holds the client secret: a `clientSecret` key, or any unknown key, fails validation.

Field	Required	Description
`label`	no	Account label, 1 to 100 characters. Defaults to the connection name.
`grantType`	yes	`authorization_code` or `client_credentials`.
`issuer`	yes	Issuer URL.

Field	Required	Description
`discoveryUrl`	unless endpoints are set	OpenID or OAuth discovery document.
`authorizationUrl`, `tokenUrl`	without `discoveryUrl`	Explicit endpoints. `authorizationUrl` is needed only for `authorization_code`.
`revocationUrl`	no	Revocation endpoint used by disconnect.
`clientId`	see below	Registered client ID.
`tokenAuthMethod`	yes	`client_secret_post`, `client_secret_basic`, or `none`. `client_credentials` cannot use `none`.
`scopes`	yes	Array of scope strings. May be empty.
`resourceOrigins`	no	Exact origins (scheme, host, port) the token may be sent to. Defaults to the origin of `baseUrl` or `url`. Not URL prefixes.
`authorizationParameters`, `tokenParameters`	no	Extra parameters. Only `access_type`, `prompt`, `audience`, `resource`, `login_hint`, and `include_granted_scopes` are accepted.
`accountUrl`, `accountIdPath`	together	Account lookup used to identify the account on reconnect. The `accountUrl` origin must be in `resourceOrigins`. `accountIdPath` is a dotted path.
`scopeSeparator`	no	`","` for a provider that separates scopes with commas. Defaults to a space.
`bearerTokenTypes`	no	`token_type` values the provider uses for a bearer token, such as Slack's `bot`.
`unreportedScopes`	no	Requested OpenID Connect scopes (`offline_access`, `openid`, `profile`, `email`) that the provider grants without listing them in the token response, such as Microsoft's `offline_access`. Each one must also be in `scopes`, and it is still requested. Every other requested scope must appear in the response.

Omit `clientId` only for a public authorization-code client: the file then needs `discoveryUrl`, `"grantType": "authorization_code"`, and `"tokenAuthMethod": "none"`, and the broker registers a client bound to the callback URL.

Hosted OAuth requires public HTTPS destinations. Local development also permits explicit loopback URLs. Resource redirects are rejected, so configure the final API origin. The broker generates protected fields such as `state` and the redirect URI.

Changing scopes, resource origins, or other compared settings in `oauth.json` moves a connected connection to reconnect required. Discovery-derived endpoints and an omitted client ID are not compared. Reconnecting approves the edited file and requires authorizing an account again.

Add a connection in Amodal

Add connection on the Connections page writes the connection files through the connected source: a commit on the selected branch for a GitHub source, a source snapshot for an Amodal source. See Where changes are saved.

The dialog stages are **Choose service**, **Set up**, **Save**, and **Connect**. Services with nothing to fill in skip **Set up**.

1. **Choose service** lists the managed providers the platform enables and the remote MCP presets, such as Linear, Jira and Confluence, Notion, GitLab, Sentry, and Stripe. A preset writes an ordinary OAuth MCP `spec.json` and needs no application registration. **Custom connection** offers an API key, your own OAuth app, or an MCP server.
2. **Set up** takes the name and the service's values. The name must match `[A-Za-z0-9_-]{1,100}` and be unique. A managed provider writes `auth: {"type": "oauth2", "provider": "<id>"}` with no `oauth.json`. A custom OAuth app shows the redirect URL to register, writes `oauth.json`, and stores the client secret encrypted outside the repository. An API key is stored as an agent secret after the files are saved.
3. **Save** shows the summary and the generated files under **Configuration files**, then commits with **Save connection**.
4. **Connect** lists the remaining steps: add the credential for each `env:` reference with **Save secret**, then **Deploy and connect** and **Continue to sign in**. Custom applications wait for the deployment to become active, because their settings come from its bundle. With manual production promotion, the dialog offers **Make production**. **Connect later** leaves **Finish setup** on the row.

An MCP server without published resource metadata, or one that requires a pre-registered client ID, needs `oauth.json`. Register the exact callback URL shown in Amodal.

Manage connections in Amodal

- Each row on the Connections page shows one state and the action that completes setup, such as **Finish setup**, **Configure OAuth**, **Add credential**, **Connect**, **Reconnect**, or **Restart agent**. A connection that declares no authentication reads **No credentials**.
- **Test connection** runs the `test` operation declared in `spec.json`, or lists an MCP server's tools. A REST connection without a test operation offers **Open** instead.
- The connection page's **Account and access** card shows the authorized account, its scopes, and **Disconnect account**. Disconnecting affects every connection bound to the same account.
- The **Settings** card of a custom OAuth connection edits `oauth.json` through the connected source. Deploy the saved source, then connect.
- When the saved account cannot be verified on reconnect, **Authorize a new account** and **Replace account and authorize** create a fresh binding. Other connections keep the replaced account.
- **Remove** deletes the directory and its OAuth binding. It stops when code or configuration still references the connection. It does not deploy, so a deployed connection stays as **Pending removal** until the next deploy. Removing a binding does not disable the account.

Credential changes do not restart a hosted runtime. New sessions resolve them immediately. **Restart agent** restarts the deployment so open sessions, channels, and MCP sessions pick them up.

Use the CLI

Commands address the connection name, such as `analytics`, not its npm package name.

`add` writes the connection files into the working tree without committing, under `amodal/connections/<name>/` when that directory holds connections, else `connections/<name>/`. It refuses an existing name. It takes one of `--rest <baseUrl>`, `--mcp <url>`, or `--provider <id>` (with an optional `--rest <baseUrl>`):

```
amodal connections add crm --rest https://api.example.com
amodal connections add shop --rest https://shop.example.com --auth header --header X-Shop-Token --
secret-name SHOP_TOKEN
amodal connections add analytics --rest https://analyticsdata.googleapis.com --auth oauth2 --oauth
./oauth.json
amodal connections add drive --provider google-drive --rest https://www.googleapis.com/drive/v3
amodal connections add linear --mcp https://mcp.linear.app/mcp
amodal connections add notion --mcp https://mcp.notion.example --auth bearer
```

`--auth` selects `bearer` (default for `--rest`), `api_key`, `basic`, `header` (with `--header <name>`), or `oauth2` (default for `--mcp`, only method for `--provider`). A token method writes `auth.token: "env:<NAME>_API_KEY"`, `<NAME>_BASIC_AUTH` for `basic`, or `<NAME>_TOKEN` for an MCP server. `--secret-name` picks another variable. `--oauth <file>` copies and validates a custom application's `oauth.json`.

The other actions are `configure`, `connect`, `status`, `test`, `list`, `disconnect`, and `remove`:

```
amodal connections configure analytics --agent AGENT_ID --config oauth.json --secret-stdin
amodal connections connect analytics --agent AGENT_ID
amodal connections status analytics --agent AGENT_ID --json
amodal connections test analytics --agent AGENT_ID --path ./agent
amodal connections list --agent AGENT_ID --json
```

- `configure` does not read the connection's `oauth.json`. Pass `--config <file>`, `--grant GRANT_ID` to bind an existing account, or `--managed PROVIDER_ID`. For a provider whose host depends on the connection, such as Egnyte, also pass `--rest` with the connection's base URL.
- `--secret-stdin` reads a confidential client's secret from standard input until it ends. Do not pass secrets as arguments.
- `connect` prints and opens the authorization link (`--no-open` skips the browser) and polls until completion or denial. The authorization expires after ten minutes. For cloud authorization, sign into the same Amodal account in the browser.
- `disconnect` disables the account, attempts revocation when the provider has a revocation endpoint, and blocks every binding of that account. It does not delete files.
- `remove` deletes the connection directory only when every file in it is a source file. With `--local` it also deletes the local binding. Without `--local`, cloud bindings and accounts are unchanged.

Local and hosted execution

Add `--local --path ./agent` to `configure`, `connect`, `status`, `test`, `list`, and `disconnect` to store credentials on this machine. The local authorization-code callback is `http://127.0.0.1:8788/callback`; `--callback-port` selects another port. Register both this URL and the platform callback shown in Amodal when one provider application serves both.

Local credentials are encrypted under `~/.amodal/oauth/` with an owner-only key file. They are not copied into the repository or the deployment bundle. A missing key requires reauthorization.

`amodal dev` uses the local bindings. `amodal dev --oauth-cloud` uses the selected cloud agent's bindings through the platform broker; select the agent with `amodal login` or `amodal agent` first.

Hosted runtimes resolve the named connection for their agent, so the connection's files must be in the active deployment. REST requests read the current binding at request time. HTTP MCP sessions bind at initialization: create a fresh session after changing an MCP application's scopes, resource origins, or account.

The `request` tool keeps its permission and confirmation checks. Custom code using `ctx.request()` must enforce its own operation authorization. Both paths enforce the approved resource origin. A failed resource request is not replayed.

Managed providers

Managed providers are OAuth applications hosted by the platform: the Google APIs below, Slack, Microsoft Outlook, and Egnyte. Select one with `auth.provider`. A provider appears in the picker only when the platform has its client ID and client secret configured.

`auth.provider`	Scopes beyond `openid`	Selection
`google-analytics`	`analytics.readonly`	Property ID
`google-sheets`	`spreadsheets`	Spreadsheet ID
`google-docs`	`documents`	Document ID
`google-drive`	`drive.file`	none
`google-calendar`	`calendar.readonly`, `calendar.events`	Calendar ID
`google-search-console`	`webmasters.readonly`	Site URL
`google-bigquery`	`bigquery`	Project ID
`egnyte`	`Egnyte.filesystem`, `Egnyte.audit`	Egnyte domain

All managed Google providers share one Google OAuth client. Drive's `drive.file` scope grants access to files created by, opened with, or shared with that Google app, per Google account. Connections that authorize the same Google account therefore see the same files, including files created through another connection. Amodal has no Google Picker integration, so it cannot authorize arbitrary existing files.

Slack

`slack` is a managed provider whose authorized account is the workspace, so reconnect requires the same workspace. The connection's `spec.json` gets the Web API base URL and `auth.test` as its test operation. The bot token has the `app_mentions:read`, `assistant:write`, `chat:write`, `im:history`, `channels:read`, `groups:read`, `channels:history`, `groups:history`, and `users:read` scopes: the agent reads the channels the app is invited to and posts as the app.

To answer direct messages and @mentions, the agent lists `@amodalai/channel-slack-assistant` among the `packages` in `amodal.json`. The package ships the `slack-assistant` channel, so the agent needs no `channels/slack-assistant/channel.json` of its own. Adding a managed Slack connection from the Studio connections page commits that `packages` entry in the same commit as the connection files, unless the package is already listed. Studio leaves `amodal.json` unchanged when the repository has none or it is not valid JSON; list the package by hand in that case.

In a channel the app is invited to, an @mention gets a reply in a thread under it. A follow-up in that thread needs another @mention, and everyone in the thread shares one conversation. Nothing else is configured per agent. The platform's Slack app receives every workspace's events at one request URL and relays each one, with the workspace's bot token, to the agent whose Slack connection is bound to that workspace. An event is not delivered when two agents are bound to the same workspace, when the grant needs reconnection, or when the agent has no active production deployment. A channel that declares its own `signingSecret` still receives relayed events, and it also accepts requests that Slack signs with that secret.

While the agent works on a message, in a direct message or a channel thread, Slack shows its working indicator with a Stop button, and the reply streams into the thread as the agent writes it. The indicator ends when the turn ends: after the reply, when the agent writes no text, when the turn fails, and when someone presses Stop. Stop ends the turn, and the text already streamed stays in the thread. Slack shows Stop only when the app subscribes to the `agent_session_stopped` bot event, which the generated manifest includes.

A custom Slack application with `"scopeSeparator": ","`, `"bearerTokenTypes": ["bot"]`, and `"accountIdPath": "team.id"` in its `oauth.json` can call the Web API, but only the managed provider receives relayed events.

Platform operators create the shared Slack app from a generated manifest. Run `node scripts/slack-app-manifest.mjs https://platform.example.com` with the platform's base URL (add `--name "My App"` to change the app name from `Amodal`). In Slack, choose Create New App, From an app manifest, and paste the output. The manifest carries the redirect URL, the Events API request URL, the event subscriptions, the Agents & AI Apps feature, and the bot scopes from the managed Slack provider. After creating the app, set `SLACK_CLIENT_ID`, `SLACK_CLIENT_SECRET`, and `SLACK_SIGNING_SECRET` on the platform from the app's Basic Information page. When `SLACK_CLIENT_ID` is set without `SLACK_SIGNING_SECRET`, the platform logs `slack_signing_secret_missing` once per process and answers every Slack event with a 503.

Microsoft Outlook

`microsoft-outlook` is a managed provider for Outlook mail and the default calendar through Microsoft Graph. Its authorized account is the Microsoft user, identified by the Graph `id` that `/me` returns, so reconnect requires the same user. The connection's `spec.json` gets the base URL `https://graph.microsoft.com/v1.0` and `GET`

`me` as its test operation. The provider requests the delegated permissions `User.Read`, `Mail.Read`, `Mail.Send`, and `Calendars.ReadWrite`, plus `offline_access` for the refresh token. The agent reads mail, sends mail and replies, manages events on the default calendar, and reads free/busy.

Platform operators register the shared application in the Microsoft Entra admin center, under App registrations, New registration:

1. Under Supported account types, choose "Accounts in any organizational directory and personal Microsoft accounts" to admit both work or school accounts and personal accounts, or "Accounts in any organizational directory" to admit work and school accounts only. A single-tenant registration fails, because the provider signs in through the `common` endpoint.
2. Add a redirect URI on the Web platform: `<platform API URL>/api/oauth/callback`.
3. Under API permissions, add the Microsoft Graph delegated permissions `offline_access`, `User.Read`, `Mail.Read`, `Mail.Send`, and `Calendars.ReadWrite`.
4. Under Certificates & secrets, create a client secret.
5. Set `MICROSOFT_CLIENT_ID` on the platform to the Application (client) ID, and `MICROSOFT_CLIENT_SECRET` to the secret's Value, not its Secret ID.

An organization that restricts user consent needs an administrator to grant consent for the tenant; until then, sign-in stops at an approval request. Publisher verification lets users consent in tenants that allow consent only for apps from verified publishers.

Client secrets expire after at most 24 months. Add a fresh secret and update `MICROSOFT_CLIENT_SECRET` before the expiry. The secret is not part of the compared configuration, so connections stay connected. Changing `MICROSOFT_CLIENT_ID` disables every Microsoft connection until it is reconnected. Microsoft ends a refresh token that goes unused for 90 days, and the connection then asks to reconnect.

Free/busy through `getSchedule` works only for work and school accounts. Microsoft has no revocation endpoint: disconnect deletes the stored tokens, and the user removes the app's access from their Microsoft account settings.

A custom Microsoft application in `oauth.json` needs explicit `authorizationUrl` and `tokenUrl` instead of `discoveryUrl`, no `openid` scope, and `"unreportedScopes": ["offline_access"]`. Microsoft's multi-tenant discovery document and ID tokens carry a per-tenant issuer that no single `issuer` matches.

Egnyte

`egnyte` is a managed provider for one Egnyte domain per connection. The domain is the name before `.egnyte.com`: for `https://baypoint.egnyte.com`, enter `baypoint`. Studio writes `https://<domain>.egnyte.com/pubapi` as the base URL in `spec.json`. The platform takes the OAuth host from that base URL and accepts only `https://<label>.egnyte.com`, where the label is lowercase letters, digits, and inner hyphens. The client secret and tokens therefore reach only that domain.

Changing the domain in `spec.json` and deploying marks the connection as changed. Reconnect then approves the new host and requires authorizing an account again.

The authorized account is the `username` that `GET /v1/whoami` returns, so reconnect requires the same Egnyte user. The test operation lists `/Shared`.

The connection requests two scopes:

- `Egnyte.filesystem` lists, reads, searches, uploads, moves, and deletes the files and folders the signed-in user can reach, and reads file system events. Egnyte has no read-only file access. Moves, copies, new folders, and deletes ask for confirmation.
- `Egnyte.audit` reads the audit stream, which records previews, downloads, and other file actions by user. It answers only for an Egnyte admin, or a power user with the "can run reports" role, on a plan with audit reporting. Other users get HTTP 403.

Access tokens last 30 days and refresh automatically. Disconnect deletes the stored tokens and does not revoke them at Egnyte; users revoke the application in their Egnyte settings. Downloads return text formats only, and uploads write the JSON request body, so the connection cannot upload text, spreadsheet, or Office files.

With the CLI, pass the base URL to both `add` and `configure`:

```
amodal connections add files --provider egnyte --rest https://baypoint.egnyte.com/pubapi
amodal connections configure files --managed egnyte --rest https://baypoint.egnyte.com/pubapi --agent
AGENT_ID
```

The CLI writes only `spec.json`. Studio also writes the policy, surface, and rules.

Platform operators register an API key at developers.egnyte.com for an application that other Egnyte domains use. Set `<platform API URL>/api/oauth/callback` as its redirect URI and request the `Egnyte.filesystem` and `Egnyte.audit` scopes. Then set `EGNYTE_CLIENT_ID` to the API key and `EGNYTE_CLIENT_SECRET` to its secret on the platform. Until Egnyte approves the key for production, only the domain registered with it can connect. Editing the provider's `config` block disables every Egnyte connection until it reconnects.

Recipes

Google Analytics 4 with your own application

Register a Google OAuth web application and enable the Analytics Data API:

```
{
  "label": "Analytics account",
  "grantType": "authorization_code",
  "issuer": "https://accounts.google.com",
  "discoveryUrl": "https://accounts.google.com/.well-known/openid-configuration",
  "clientId": "YOUR_CLIENT_ID",
  "tokenAuthMethod": "client_secret_post",
  "scopes": ["openid", "https://www.googleapis.com/auth/analytics.readonly"],
  "resourceOrigins": ["https://analyticsdata.googleapis.com", "https://openidconnect.googleapis.com"],
  "authorizationParameters": { "access_type": "offline" },
  "accountUrl": "https://openidconnect.googleapis.com/v1/userinfo",
  "accountIdPath": "sub"
}
```

`openid` and the account lookup identify the account on reconnect; the broker does not use the ID token as the user's identity. `access_type: offline` obtains a refresh token. Add `prompt: "consent"` only when reconsent is needed.

Test against a real property. For property `123456789`:

```
{
  "baseUrl": "https://analyticsdata.googleapis.com/v1beta",
  "auth": { "type": "oauth2" },
  "test": {
    "method": "POST",
    "path": "properties/123456789:runReport",
    "body": {
      "dateRanges": [{ "startDate": "yesterday", "endDate": "yesterday" }],
      "metrics": [{ "name": "eventCount" }],
      "limit": "1"
    }
  }
}
```

An empty successful report verifies access. An account without access to the property fails the test even when OAuth connects.

GitLab

```
{
  "label": "GitLab account",
  "grantType": "authorization_code",
  "issuer": "https://gitlab.com",
  "discoveryUrl": "https://gitlab.com/.well-known/openid-configuration",
  "clientId": "YOUR_CLIENT_ID",
  "tokenAuthMethod": "client_secret_post",
  "scopes": ["read_api", "read_user"],
  "resourceOrigins": ["https://gitlab.com"],
  "accountUrl": "https://gitlab.com/api/v4/user",
  "accountIdPath": "id"
}
```

With base URL `https://gitlab.com/api/v4`, use `GET user` as the test.

Auth0 client credentials

Create a machine-to-machine application authorized for the API, with token authentication set to Client Secret (Post):

```
{
  "label": "Reporting service",
  "grantType": "client_credentials",
  "issuer": "https://YOUR_TENANT.us.auth0.com/",
  "tokenUrl": "https://YOUR_TENANT.us.auth0.com/oauth/token",
  "clientId": "YOUR_CLIENT_ID",
  "tokenAuthMethod": "client_secret_post",
  "scopes": ["read:reports"],
  "resourceOrigins": ["https://YOUR_API_HOST"],
  "tokenParameters": { "audience": "YOUR_API_IDENTIFIER" }
}
```

The audience is the Auth0 API identifier, and the resource origin is the API host. This grant has no account lookup or browser consent; expired tokens are reacquired with client credentials.

Troubleshooting

- Revoked credentials or reduced consent: reconnect the account.
- Configuration revision errors: approve the intended `oauth.json` by reconnecting.
- Resource permission failures: grant the account access to the selected resource.
- An approved scope that returns 403: enable the matching API in the provider's project.

Error messages include safe codes and correlation IDs. Token values never appear in diagnostics.

AGENT REPO

Skills

A skill is a Markdown workflow the agent can load when a task needs it. Use skills for investigation steps, decision criteria, and response requirements. Put calculations and required execution order in tools or workflows.

```
amodal/skills/
├── triage/
│   └── SKILL.md
└── account-access/
    └── SKILL.md
```

SKILL.md Format

Heading-based

```
# Skill: Account Access Triage

Check account-access tickets and recommend the next support action.
Trigger: When the user asks about login failure, password reset, or account lock.

## Behavior

1. Load the ticket and identify the affected account.
2. Load the account-access policy with load_knowledge.
3. Check whether identity verification is complete.
4. If verification is missing, recommend verification before account changes.
5. Return the ticket status, confirmed facts, missing information, and next action.

## Constraints

- Do not request passwords or credentials.
- Do not claim a reset succeeded unless the account system confirms it.
```

Use the parsed name, `Account Access Triage`, when referencing this skill in an agent's `skills` list. The directory name does not override it.

Frontmatter-based

Use frontmatter for a stable machine-readable name, eager loading, or tool declarations. `name` is required; a frontmatter file without it is not loaded.

```
---
name: account-access
description: Check account-access tickets and recommend the next support action.
trigger: When the user asks about login failure, password reset, or account lock.
load: on-demand
allowedTools: [lookup_ticket, load_knowledge]
---

1. Load the ticket and the account-access policy.
2. Check identity-verification status before recommending changes.
3. Return confirmed facts, missing information, and the next action.
```

Parsed Fields

Field	Source	Meaning
`name`	`# Skill: Name` or frontmatter	Name used by `load_skill` and agent resource declarations.
`description`	Text before the first `##` section, excluding `Trigger:`, or frontmatter	Short description included in the skill index. Heading-format files without a `##` section have an empty description.
`trigger`	One unwrapped `Trigger:` line or frontmatter	Tells the model when the skill applies.
`body`	From the first `##` section onward, or the content after frontmatter	Instructions returned by `load_skill`.
`load`	Frontmatter	`on-demand` (default) or `eager`. Other values warn and use the default.
`allowedTools`	Frontmatter	Tool names added to a root agent's scope when it declares this skill.
`requires`	Frontmatter	`{ env?: string[], bins?: string[] }` eligibility requirements checked before exposing the skill.
`resultTool`	Frontmatter	Tool name whose output completes the skill.

Choose one format per file. A `# Skill:` heading takes precedence over frontmatter, so combining them prevents frontmatter-only options from being read. A skill directory can also include supporting subdirectories (`files/`, `scripts/`, `references/`, `assets/`) and an optional `sandbox-setup.sh`, which are copied into the session's sandbox when the skill is loaded.

Skill Activation

A root session's prompt lists its available skills by name, trigger, and description. The model calls `load_skill` to retrieve an on-demand body. `load: eager` includes the body in the initial prompt.

A named agent exposes only the skills in its `skills` list. Unscoped chat exposes the bundle's skills. If `agents/default/` exists, an omitted chat `agent` selects that scoped agent. A delegated subagent uses its own prompt and declared tools; include `load_skill` in its `tools` if it needs to load skills available through the parent registry.

A trigger describes when to use a skill; it does not schedule work or guarantee activation. Verify activation with a `skill_loaded:` assertion in an eval.

The model can load several skills in one conversation. To chain them, name the next skill in the instructions, such as "If the ticket also disputes a charge, load billing-review". Skill instructions do not enforce order or permissions; use a composite tool for a fixed sequence and enforce approval in code. Keep shared facts in knowledge.

AGENT REPO

Knowledge Base

Knowledge is reference material for the agent, such as policies, terminology, and procedures, kept in `amodal/knowledge/`. It takes three forms:

- A Markdown document, loaded whole.
- A **corpus**, a searchable collection committed with the agent.
- A **library**, a searchable collection your application files documents into while the agent runs, scoped per document.

Use stores for changing structured records and connections for live API access. Knowledge reaches the model, so do not put secrets in it.

Document Format

Small documents are Markdown files:

```
amodal/knowledge/
├── support-policy.md
└── product-glossary.md
```

```
---
description: Use when a support ticket involves account access or a disputed charge.
category: support
tags: [account-access, billing]
---

# Support Policy

- Check account-access verification before recommending account changes.
- Confirm a refund's status in the billing system before reporting it as complete.
- Escalate a disputed charge to the billing team with the invoice and ticket IDs.
```

The filename without `.md` is the document name: `support-policy`. The title comes from frontmatter `title`, then the first `#` heading, then the filename. Optional frontmatter includes `description`, `category`, and `tags`.

A root session's prompt lists document names and descriptions. The model retrieves a body with:

```
load_knowledge({ knowledge: 'support-policy' });
```

Without frontmatter `description`, the body's first paragraph is used, truncated to 400 characters. Write the description to say when the document applies.

A delegated subagent receives its authored prompt rather than the root's compiled knowledge index. Declare `load_knowledge` in the subagent's `tools`; the tool description lists available documents.

Searchable Corpora

A corpus is a directory with a `knowledge.json` manifest beside its pages:

```
amodal/knowledge/reference-manual/
├── knowledge.json
├── _SECTIONS.tsv
└── pages/
    ├── 01-overview.md
    └── 02-procedures.md
```

```
{
  "type": "corpus",
  "title": "Reference Manual",
  "description": "Operating procedures and section references.",
  "pages": "pages",
  "index": "_SECTIONS.tsv",
  "provenance": {
    "edition": "2026-01",
    "source": "https://example.com/manual"
  }
}
```

`pages` defaults to `pages`. The optional TSV `index` has `section`, `title`, `page`, and `line` columns, with a header row. Without it, citations identify pages and lines instead of sections.

`knowledge_search` has `sections`, `files`, `content`, and `documents` modes, plus `hybrid` when the corpus has a semantic index. `knowledge_read` reads a section or a line range. A named agent must declare both tools in its `tools`. Build a semantic index with `amodal corpus index`.

Runtime Libraries

A library uses the same retrieval as a corpus, over documents your application files at runtime. Declare it with a manifest and no pages:

```
amodal/knowledge/team-library/
└── knowledge.json
```

```
{
  "type": "library",
  "title": "Team library",
  "description": "SOPs and agreements operators filed for the agent to consult.",
  "scope": { "keys": ["mission_id"] },
  "parsing": { "preference": ["docling", "builtin"] }
}
```

`scope.keys` names the scope context keys a document may be scoped by. A document filed with `{ "mission_id": "m1" }` is visible only to sessions that carry that value. A document filed with no scope is visible to every session. Search runs only over the visible set.

`scope.source` sets where a session's values come from. The default, `claims`, reads only the verified token claims, so a caller cannot widen its view through the request body. Set `"source": "context"` only when your application enforces access itself and passes the current scope in the request body.

The document routes require an API key or a token minted with `role: "operator"`. End-user chat tokens are refused; users reach a library only through the agent's tools, under their own scope.

Filing documents

The runtime parses, chunks, embeds, and indexes the document. Your application keeps the original file.

```
POST /api/knowledge/team-library/documents
Authorization: Bearer <runtime token>

{
  "doc_id": "doc-a1b2c3",
  "title": "Retrans SOP",
  "filename": "retrans-sop.pdf",
  "mime_type": "application/pdf",
  "bytes_base64": "...",
  "scope": { "mission_id": "m1" },
  "tags": ["sop"],
  "parser": "auto"
}
```

- Send `text` in place of `bytes_base64` when you already hold the text.
- The response's `document` carries `status` (`ready`, or `parsing` with status `202` while a service parser runs), `parser`, `outline`, `chunks`, and `warnings`.
- `GET /api/knowledge/:library/documents` lists documents. `GET .../documents/:id?text=1` returns the parsed text. A `GET` also advances a pending parse.
- `PATCH .../documents/:id` changes `title`, `description`, `scope`, `tags`, `metadata`, and `enabled` without re-parsing. `enabled: false` hides the document from the agent.
- `DELETE .../documents/:id` retires the document.

- To re-parse with another parser, `POST` the same `doc_id` with `parser` set.

Parsers

`GET /api/knowledge/parsers` lists the parsers a deployment can use:

Parser	Kind	Configured by
`builtin`	local	Always available: PDF text layer, DOCX, XLSX, Markdown, text.
`docling`	service	`DOCLING_URL` (a docling-serve instance), optional `DOCLING_API_KEY`.
`textract`	service	`AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, `AWS_REGION`; `TEXTRACT_S3_BUCKET` for multi-page PDFs.
`google-document-ai`	service	`GOOGLE_DOCUMENT_AI_PROCESSOR` plus `GOOGLE_APPLICATION_CREDENTIALS_JSON` or `GOOGLE_DOCUMENT_AI_ACCESS_TOKEN`.
`azure-document-intelligence`	service	`AZURE_DOCUMENT_INTELLIGENCE_ENDPOINT`, `AZURE_DOCUMENT_INTELLIGENCE_KEY`.

Without a `parser` in the request, the runtime tries the library's `parsing.preference`, then `KNOWLEDGE_PARSER_PREFERENCE`, then `builtin`. A service failure falls through to the next parser and adds a warning to the document. A scanned PDF with no text layer fails `builtin` and needs an OCR-capable service.

What the model sees each turn

A corpus is described once in the cached prompt. A library is described on every turn, after the cache breakpoint, as the list of documents the session can see with their titles and descriptions. `retrieval` with `auto: true` runs one hybrid search with each user message and shows the top `hits` (1 to 10, default 3) to the model:

```
{
  "type": "library",
  "scope": { "keys": ["mission_id"] },
  "retrieval": { "auto": true, "hits": 3 }
}
```

`auto` defaults to `false`, so retrieval does not run without it. Automatic retrieval costs a search and a few hundred tokens per turn.

Inspecting what the runtime holds

- `GET /api/knowledge` lists every corpus and library with its embedding settings, whether it is indexed, and, for a library, document counts by status. Its `markdown` array lists the Markdown documents by name, title, and description.
- `GET /api/knowledge/:library/search?q=&mode=&scope=` runs the agent's `knowledge_search`. Without `scope` it searches every document. With `scope` set to a session's scope context as JSON, it returns what that session would see.

Both routes use the operator authentication of the filing routes.

Storage

Hosted runtimes store library documents on the platform with no configuration. A self-run runtime uses `CORPUS_DATABASE_URL`, or `DATABASE_URL`, with pgvector. Set `"embedding": { "store": "memory" }` for an in-process store during local development.

One request carries one document, so parsed text is limited to a few megabytes. Without `CORPUS_EMBEDDING_URL`, a library is searchable by words only, and each document's `warnings` say so.

AGENT REPO

Stores

Stores are typed collections the agent reads and writes during sessions. Each file in `amodal/stores/*.json` defines one store, and the runtime generates its tools.

Store Definition

```
{
  "name": "research-notes",
  "entity": {
    "name": "ResearchNote",
    "key": "{company_id}:{topic}",
    "schema": {
      "company_id": { "type": "string" },
      "topic": { "type": "string" },
      "summary": { "type": "string" },
      "confidence": { "type": "number", "min": 0, "max": 1 },
      "updated_at": { "type": "datetime" },
      "tags": { "type": "array", "item": { "type": "string" } },
      "metadata": {
        "type": "object",
        "fields": {
          "source": { "type": "string" },
          "analyst": { "type": "string", "nullable": true }
        }
      }
    }
  },
  "history": { "versions": 3 }
}
```

Field	Description
`name`	Lowercase letters, digits, `-`, and `_`, starting with a letter. Defaults to the file name; a different name is a load error.
`entity.name`	Entity type name.
`entity.key`	Key template built from document fields, such as `{company_id}:{topic}`.
`entity.schema`	Field definitions. A flat `key` and `schema` at the top level are also accepted in place of `entity`.
`history.versions`	Number of previous versions kept per document.

Field	Description
`ttl`	Expiry in seconds, or `{ "default": 86400, "override": [{ "condition": "status = 'done'", "ttl": 3600 }] }`. Conditions support `field = 'v'`, `field != 'v'`, and `field IN ['a', 'b']`.
`deletable`	`true` registers the `remove` tool. Default `false`.
`shared`	`true` makes one collection readable by every scope. See Scoped vs. Shared Stores.
`indexed`	Array of schema field names indexed for equality filtering.
`failure`	`"ThrowError"` (default) or `"ReturnEmpty"` on store read errors.
`trace`	`"Full"` (default), `"Redacted"`, or `"None"` for store span attributes.

Field Types

Type	Extra fields
`string`	`nullable`
`number`	`min`, `max`, `nullable`
`boolean`	`nullable`
`datetime`	`nullable`
`enum`	`values` (required)
`array`	`item` (required)
`object`	`fields` (required)
`ref`	`store` (required)

Every type accepts `nullable`.

Generated Tools

For a store named `research-notes`, the runtime registers `store__research_notes__get`, `__set`, `__query`, `__list`, and, with `"deletable": true`, `__remove`. Hyphens in the store name become underscores. See Built-in Tools.

Custom tools write with `ctx.store(name, payload)`, which resolves the key from the payload.

Per-Agent Access

A named agent lists the stores its sessions can use, with `"read"` or `"rw"` access:

```
// agents/research/agent.json
{
  "name": "Research",
  "stores": {
    "company-profiles": "read",
    "research-notes": "rw"
  }
}
```

`read` grants `get`, `query`, and `list`. `rw` adds `set` and `remove`.

Storage Backend

Hosted deployments supply the backing database. `amodal dev` uses an in-memory store, so local data is lost when the dev server restarts.

Scoped vs. Shared Stores

Store data is partitioned by the request's `scope_id`; the hosted runtime does not require one (see Scope). `"shared": true` makes one collection that every scope reads:

```
{
  "name": "company-profiles",
  "shared": true,
  "entity": {
    "name": "CompanyProfile",
    "key": "{company_id}",
    "schema": {
      "company_id": { "type": "string" },
      "name": { "type": "string" }
    }
  }
}
```

The runtime rejects writes and deletions to a shared store, including when an agent declares `"rw"`. Populate reference catalogs outside the agent, and use scoped stores for records the agent updates.

AGENT REPO

Tools

Built-in tools cover connection calls, stores, knowledge, skills, and chat controls. Custom tools in `amodal/tools/<name>/` run your own code. A custom tool takes one of two layouts:

```
amodal/tools/
├── summarize_ticket/
│   └── tool.ts            ← metadata and handler in one module
└── review_request/
    ├── tool.json          ← metadata
    ├── handler.ts         ← default export (params, ctx) => result
    ├── package.json       ← (optional) npm dependencies
    └── requirements.txt   ← (optional) Python dependencies
```

When `tool.ts` exists, the directory uses that layout and ignores any `tool.json`. Composition (`ctx.callTool`, `ctx.callSubagent`), durable execution, triggers, the `env` allowlist, and UI labels require the `tool.json` layout.

Tool code ships with the agent repository. Commit, deploy, and read the build logs for TypeScript, dependency, and validation errors. See Deployment API.

Built-in Tools

Tool	Registered when	What it does
`request`	≥1 REST connection without `openapi` or `skipRequestTool: true`	HTTP calls with connection authentication and policy checks. See Confirmation Model.
`<connection>__<operationId>`	connection with `openapi.source`	Typed operation calls. See Native OpenAPI connections.
`<connection>__discover`	connection with `openapi.source`	Search and activate operation tools for the next reply.
`read_spilled_result`	≥1 native operation tool	Page a large stored connection result.
`store__<store>__get`	store defined	Read one document by key. Hyphens in the store name become underscores. See Stores.
`store__<store>__set`	store defined	Create or update one document.
`store__<store>__query`	store defined	Filter documents.
`store__<store>__list`	store defined	List documents.
`store__<store>__remove`	store sets `deletable: true`	Delete one document.

Tool	Registered when	What it does
`call_subagent`	≥1 invokable subagent	Run authored subagents, up to 12 per call and five at a time. See Agents.
`present`	unscoped chat, or declared in a named agent's `tools`	Render a widget such as an info card, data table, or metric.
`stop_execution`	always	End the current turn.
`ask_choice`	always	Ask a single- or multi-select question. Calling it ends the turn; the choice arrives as the next user message.
`load_skill`	≥1 on-demand skill in scope	Return a skill's instructions.
`load_knowledge`	≥1 Markdown knowledge document	Return a knowledge document's body.
`knowledge_search`	≥1 corpus or library, and tool in scope	Search a corpus or library.
`knowledge_read`	≥1 corpus or library, and tool in scope	Read a section or line range.
`web_search`, `fetch_url`	`webTools` configured and tool in scope	See Web Tools.
`memory`	memory enabled and editable, and tool in scope	Persistent memory with `add`, `remove`, `list`, and `search`. See Memory.
`*_repo_file`, `*_repo_files`	declared in the agent's `tools`	`read_repo_file`, `write_repo_file`, `edit_repo_file`, `delete_repo_file`, `list_repo_files`, `glob_repo_files`, `grep_repo_files`, `read_many_repo_files`. Writes, edits, and deletes require confirmation. Disable all with `"fileTools": false` in `amodal.json`.
`bash`, `read_file`, `write_file`, `publish_file`	`sandbox.provider` configured in `amodal.json` and tool in scope	Run shell commands and read, write, or publish files in the session's isolated sandbox. See amodal.json.

Tool	Registered when	What it does
`propose_source_change`	hosted, declared in the agent's `tools` (or `agents/main/agent.json` for default chat)	Propose edits to the agent's own prompt, skills, knowledge, and tool code. They wait in Studio under Proposals; approval commits and deploys them. See Proposals. Pair it with `list_repo_files` and `read_repo_file`. Registering it also registers `read_amodal_docs`, which reads the Amodal docs index or one docs page. An agent cannot add a new tool or change `amodal.json`, `agent.json`, `tool.json`, connections, or secrets.
`<server>__<tool>`	≥1 MCP server connected	Tools discovered from each MCP server.

A tool is "in scope" in unscoped chat, or when a named agent lists it in `tools`. In default chat, file tools come from the tools declared by `agents/main/agent.json`.

A skill added through `propose_source_change` reaches a named agent that lists its skills in `agent.json` only after someone adds it to that list.

Web Tools

```
{
  "webTools": {
    "provider": "google",
    "apiKey": "env:GOOGLE_API_KEY"
  }
}
```

`"webTools": true` is shorthand for the Google provider with `env:GOOGLE_GENERATIVE_AI_API_KEY`. `google` is the only provider. Both tools use Gemini grounding regardless of the agent's model. Set `model` to a Gemini model that supports Search and urlContext grounding, or omit it for the runtime default.

- `web_search({query, max_results})` returns a synthesized answer with cited source URLs. `max_results` defaults to 5, maximum 10.
- `fetch_url({url, prompt})` fetches and summarizes a public page through urlContext, and falls back to a local fetch with Readability extraction. The local fetch has a 10-second timeout and a 1 MB body cap. Each hostname is limited to 10 requests per 60 seconds.
- Under `amodal dev`, private-network URLs (localhost, RFC 1918, `.local`) use the local fetch. A hosted runtime refuses private-network and other non-public addresses.

Single-file Tools

`amodal/tools/<name>/tool.ts` holds the metadata and the handler:

```
export default {
  id: 'summarize_ticket',
  exposure: { kind: 'open' },
  base: {
    name: 'summarize_ticket',
    description: 'Load a support ticket and return a concise operator summary.',
    parametersJsonSchema: {
      type: 'object',
      properties: { ticket_id: { type: 'string' } },
      required: ['ticket_id'],
    },
  },
  async handle(ctx) {
    const { ticket_id } = ctx.input;
    const ticket = await ctx.request('support-api', `/tickets/${encodeURIComponent(ticket_id)}`);
    const needsVerification = ticket.category === 'account_access' && ticket.identity_verified !== true;
    return {
      ticket_id: ticket.id,
      status: ticket.status,
      needs_verification: needsVerification,
    };
  },
};
```

The default export must have a `handle(ctx)` function. Parameters arrive in `ctx.input`.

Field	Default	Description
`id` (or `base.name`)	directory name with `-` → `_`	Tool name. Must be snake_case.
`base.description`	tool name	Shown to the model. Say when to use the tool.
`base.parametersJsonSchema`	`{"type":"object","properties":{}}`	JSON Schema for the input.
`exposure.kind`	`open`	`open` runs without approval and is read-only for `ctx.request`. `requires-confirmation` pauses every model-selected call for approval. `operator-gated` behaves like `open`.
`enabled`	`true`	`false` skips registration.
`llm_callable`	`true`	`false` hides the tool from the model. Composites and triggers can still call it.

A `tool.ts` tool has a fixed 30-second timeout and an empty `env` allowlist, so `ctx.env()` returns `undefined`.

Tool Metadata

`tool.json` holds the metadata, and `handler.ts` default-exports `(params, ctx) => result`:

```
{
  "description": "Classify support ticket priority from category, impact, and age.",
  "parameters": {
    "type": "object",
    "properties": {
      "category": { "type": "string" },
      "affected_users": { "type": "number" }
    },
    "required": ["category", "affected_users"]
  },
  "timeout": 10000,
  "env": ["PRIORITY_RULES_URL"],
  "runningLabel": "Classifying {{category}} ticket",
  "completedLabel": "Classified ticket"
}
```

```
export default async function (params, ctx) {
  const broad = params.affected_users >= 10;
  return { priority: params.category === 'account_access' && broad ? 'high' : broad ? 'medium' :
'normal' };
}
```

The tool name is the directory name, which must be snake_case.

Field	Default	Description
`description`	required	Shown to the model.
`name`	directory name	Must match the directory name when set.
`parameters`	`{}`	JSON Schema for the input.
`confirm`	`false`	`true` or `"review"` lets `ctx.request` send write-intent calls. Neither prompts the user. `"never"` skips registration with a warning.
`timeout`	`30000`	Handler timeout in milliseconds (`0` disables the overall deadline for a durable tool).
`enabled`	`true`	`false` skips registration. The model, composites, and triggers cannot call the tool.
`llm_callable`	`true`	`false` hides the tool from the model. Composites and triggers can still call it.

Field	Default	Description
`internal`	`false`	`true` marks the tool's calls as internal plumbing, which the chat widget hides unless `verboseTools` is set.
`env`	`[]`	Environment variables `ctx.env()` may read.
`runningLabel`, `completedLabel`		Chat card text while running and after success. `{{param}}` substitutes a call parameter.
`displayName`		Noun phrase naming the tool in aggregating UIs.
`reminder`, `reminderAfterTurns`	`3`	Text restated to the model after the tool goes unused for that many turns.
`uses`, `execution`, `triggers`		Composition, durable execution, and non-model triggers. See Agent Workflows.

The loader accepts `responseShaping` and `sandbox` in `tool.json` but does not apply them. A `tool.json` tool cannot require approval: use `tool.ts` with `exposure.kind: "requires-confirmation"` for that, or gate the action inside a durable tool.

Handler Context

Member	Description
`ctx.request(connection, endpoint, options?)`	Authenticated call through a connection. `options` takes `method`, `params`, `data`, and `intent`.
`ctx.exec(command, options?)`	Run a shell command. `options` takes `cwd` and `timeout`. Resolves to `{stdout, stderr, exitCode}`.
`ctx.store(storeName, payload)`	Write a document to a store. Resolves to `{ key }`.
`ctx.env(name)`	Read an allowlisted environment variable. `undefined` when not allowlisted.
`ctx.log(message)`	Write a log line.
`ctx.signal`	`AbortSignal` that fires on cancellation or timeout.
`ctx.scopeId`	Per-user scope key for the turn. Empty string at agent level.
`ctx.scopeContext`	The turn's scope context, as sent by the embedding app. It can carry client-supplied values, so do not use it for authorization. `undefined` when absent.
`ctx.attachments`	Files attached to the turn's message. Top-level tools only.
`ctx.stageAttachments(paths?)`	Copy turn attachments into the session sandbox and return their sandbox paths.
`ctx.setLabel(running, completed?)`	Update the tool call's running and completed labels in chat (`tool_label_update`).

Member	Description
`ctx.emitReasoning(text)`	Stream reasoning text during tool execution.
`ctx.emit(widgetType, data)`	Emit an inline widget event from a `tool.ts` handler.
`ctx.artifacts`	`providers` list and `register({ providerId, sourceRef, idempotencyKey })` for registering verified app-stored artifacts when `artifactProviders` are configured.
`ctx.callTool`, `ctx.callSubagent`	Composition. See Agent Workflows.

Durable tools also get `ctx.requestInput`, `ctx.waitForApproval`, `ctx.sleepUntil`, `ctx.step`, `ctx.now`, `ctx.random`, `ctx.setDeadline`, and `ctx.artifactJobs`. See Durable Tools.

`ctx.request` rules

`ctx.request` resolves the connection's base URL, attaches its credentials and applies the connection's field restrictions to the response. It refuses an endpoint with a dot segment, a backslash or a control character. It does not evaluate `policy.json` permissions or confirmation, and it does not retry. Enforce authorization in the tool or the downstream API.

The call's intent defaults to `write` for `POST`, `PUT`, `PATCH`, and `DELETE`, and to `read` otherwise. A read-only tool (`exposure.kind: "open"` in `tool.ts`, or `confirm: false` in `tool.json`) throws on write-intent calls. Pass `intent: 'read'` for a read-only POST, such as a search endpoint:

```
const report = await ctx.request('analytics', 'properties/123:runReport', {
  method: 'POST',
  intent: 'read',
  data: { metrics: [{ name: 'eventCount' }] },
});
```

Naming

Tool names must be snake_case: lowercase letters, digits, and underscores, starting with a letter.

AGENT REPO

Agent Workflows

A composite tool calls other tools and subagents in code. Use it when an operation needs a fixed sequence, parallel checks, or a structured result. Durable execution adds recorded steps and pause and resume. Use a single tool for a self-contained calculation or API action.

Composite Tools

Use `handler.ts` with `tool.json`, and declare composition in `tool.json`:

```
amodal/tools/review_request/
├── tool.json
└── handler.ts
```

```
{
  "name": "review_request",
  "description": "Load a request, review completeness and policy, and save the findings.",
  "parameters": {
    "type": "object",
    "properties": { "request_id": { "type": "string" } },
    "required": ["request_id"]
  },
  "uses": {
    "tools": ["load_request", "persist_review_findings"],
    "subagents": ["prescreen-reviewer", "policy-reviewer"]
  }
}
```

```
export default async function (params, ctx) {
  const request = await ctx.callTool('load_request', {
    request_id: params.request_id,
  });
  const [completeness, policy] = await Promise.all([
    ctx.callSubagent('prescreen-reviewer', 'Check field completeness.', request),
    ctx.callSubagent('policy-reviewer', 'Review policy requirements.', request),
  ]);
  const findings = { completeness, policy };
  await ctx.callTool('persist_review_findings', {
    request_id: params.request_id,
    findings,
  });
  return { request_id: params.request_id, findings };
}
```

The example requires the two named child tools and two authored subagents. Subagent results are text. Parse and validate them in a child tool if your product needs structured fields.

Do not add `tool.ts` to this directory. A `tool.ts` takes precedence over `tool.json`, and that layout does not bind `ctx.callTool` or `ctx.callSubagent`.

Composition Semantics

- Declare every child tool in `uses.tools` and every specialist in `uses.subagents`. A declared name that does not resolve fails at bundle load. A call to an undeclared name fails at run time.
- `ctx.callTool(name, params)` executes the child tool and returns its result.
- `ctx.callSubagent(name, task, input?, opts?)` runs the specialist's agent loop and returns its final text. `input` is appended to the task as a JSON block.
- A failed inline child call throws a `CompositeCallError` with a `code`: `depth-exceeded`, `undeclared`, `unknown-tool`, `confirmation-unsupported`, `invalid-params`, `execution-failed`, `subagent-undeclared`, `subagent-unknown`, or `subagent-failed`.
- Inline `ctx.callTool` nesting stops at depth 4. Subagent nesting stops at five levels.
- Nested tool calls render as nested cards in the chat UI.

Nested tool calls do not run the outer loop's permission or hook checks. Inline composites refuse tools marked `requiresConfirmation` and connection-category tools, including `request`, with `confirmation-unsupported`. They do not open an approval dialog.

Durable child execution does not apply that confirmation check. Use an explicit durable approval before a write that needs an operator decision. A `uses` declaration limits reachability. Child code and downstream APIs must enforce authorization.

Fanning Out Attachments

Top-level handlers receive the current turn's uploads in `ctx.attachments`. Pass selected files in the fourth argument to a specialist:

```
export default async function (params, ctx) {
  const results = await Promise.all(
    (ctx.attachments ?? []).map((file) =>
      ctx.callSubagent(
        'document-extractor',
        "Extract the document's facts and cite the supporting text.",
        { filename: file.filename },
        { attachments: [file] },
      ),
    ),
  );
  return { results };
}
```

Declare `document-extractor` in `uses.subagents`. Attachments become file parts on the specialist's user turn. For providers without media support, they are dropped and the runtime logs `subagent_attachments_stripped`.

Durable Tools

Use durable execution when a tool must pause for input, approval, or a future time. The handler runs again from the top on resume. Journaled calls return their recorded results instead of executing again.

Set `"execution": "durable"` in `tool.json`:

```
{
  "name": "release_to_environment",
  "description": "Ask for a target environment and release to it.",
  "execution": "durable",
  "parameters": { "type": "object", "properties": {} },
  "uses": { "tools": ["run_release"] }
}
```

```
export default async function (params, ctx) {
  const target = await ctx.requestInput({
    question: 'Where should this release run?',
    inputType: 'choice',
    options: [
      { value: 'staging', label: 'Staging', recommended: true },
      { value: 'production', label: 'Production', sub: 'Makes the release live' },
    ],
  });
  const result = await ctx.callTool('run_release', { target: target.value });
  return { target: target.value, result };
}
```

The example requires a `run_release` child tool that validates the target and performs the release.

Method	Contract
`ctx.requestInput(opts)`	Pause for an answer. Supports `question`, `context`, `inputType`, `options`, `payload`, `allowEdits`, `eligibleAnswerers`, and `excludeAnswerers`. Returns `{value, respondedBy, edits?}`.
`ctx.waitForApproval(prompt)`	Yes/no form of `requestInput`. Returns `{approved, respondedBy?}`.
`ctx.step(name, fn)`	Run an async checkpoint and record its JSON result. On replay, return that result without calling `fn`. Repeated names are distinguished by call order.
`ctx.sleepUntil(when)`	Pause until a `Date`, then resume through the scheduler.
`ctx.now()`	Recorded clock value in milliseconds.
`ctx.random()`	Recorded random value.
`ctx.setDeadline(ms)`	Extend or replace the overall durable run deadline in milliseconds (`0` disables the overall deadline).
`ctx.artifactJobs.run(...)`	Run a multi-stage durable artifact job with checkpointed `job.stage(opts, fn)` steps, stage deadlines, progress reporting, and output verification.

`inputType` accepts `text` (the default), `choice`, or `yesno`. Choice options have a `value` and `label`. `recommended` and `sub` affect presentation. The returned `value` is whatever the answerer posted, so validate it before a sensitive action. `eligibleAnswerers` and `excludeAnswerers` are lists of user IDs.

Use `ctx.step` for handler-local work whose result must survive replay:

```
const record = await ctx.step('parse-input', async () => JSON.parse(params.document_json));
```

Return plain data from checkpoints. A `Date` returns as an ISO string, and a class instance returns as a plain object. Mutations to variables outside `fn` are not restored on replay.

Route external I/O through declared child tools. In a durable handler, `ctx.request`, `ctx.store`, and `ctx.exec` reject, and `ctx.env` returns undefined. The handler runs again from the top on each resume, so any work outside journaled calls runs again.

When a durable tool pauses during chat, the model receives `{durable: true, status: "running", session_id, message}` and the turn continues. When the tool is waiting on `requestInput`, the result is `{durable: true, status: "awaiting_operator", session_id, question, context, message}`. The run's status is `awaiting-input`. Persistence across restarts requires a persistent session store. Local `amodal dev` keeps sessions in memory.

Answering A Pause

Answer the durable session named by `session_id` in the pause result. Its ID differs from the chat session's ID.

```
curl "$RUNTIME/sessions/pending-approvals" \
  -H "Authorization: Bearer $RUNTIME_TOKEN"

curl -X POST "$RUNTIME/api/sessions/$DURABLE_SESSION_ID/answer" \
  -H "Authorization: Bearer $RUNTIME_TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"value": "staging"}'
```

The body requires `value`. With `allowEdits`, include `edits` with the edited payload. A completed run returns `200` with `outcome` and `result`. A run that pauses again returns the next pause in `outcome`.

Status	`error`	Cause
`400`	`missing_value`	The body has no `value`.
`403`	`not_eligible_answerer`	The caller is not in `eligibleAnswerers`.
`403`	`excluded_answerer`	The caller is in `excludeAnswerers`.
`404`	`session_not_found`	No such session, or it belongs to another scope.
`409`	`session_not_awaiting_input`	The pause was already answered. The body includes `answeredBy` if known.

`POST /api/sessions/:id/cancel` stops a parked run. Answer and cancel publish `session_updated` events on `GET /api/events`.

In the React SDK, `useSessionActions(id).answer(value, edits?)` answers from an inbox or custom screen. The chat widget renders inline choice, yes/no, and text cards and answers them through the same endpoint.

Triggers

Entry point	Use
LLM tool selection	The model chooses the tool in response to a user message.

Entry point	Use
Predicate trigger	A `regex` or `predicate` entry in the tool's `triggers` starts the tool from a matching message without model selection.
Direct invocation	Your app calls `POST /api/tools/:name/run` for a durable tool whose `triggers` include `{ "kind": "invoke" }`.
Automation	A platform automation runs the tool on a schedule or event.

`tool.json` accepts a `{ "kind": "schedule", "cron": "..." }` trigger, but the runtime does not fire it. Schedule the tool through an automation.

A trigger does not remove the tool from the model's tool list. Set `"llm_callable": false` in `tool.json` or `tool.ts` to hide the tool from the model while composites and triggers can still call it.

`POST /api/tools/:name/run` takes `{"input": {...}}` and optional `session_id`, `scope_id`, and `context`. It returns `403 tool_not_invocable` without an `invoke` trigger and `400 tool_not_durable` for an inline tool. It does not validate `input` against a `tool.json` parameter schema, so the handler must validate it.

Before deploying, cover denied approval, missing data, child-tool failure, and replay after a pause with evals.

AGENT REPO

Hooks

Hooks inspect or modify messages and tool calls in every agent loop: the root chat loop and each subagent loop it starts. Use them for input checks, output redaction, and approval rules on model-selected calls.

`preToolUse` and `postToolUse` run in every agent loop. That covers the root chat loop and each subagent loop, whether `call_subagent`, a composite or durable tool's `ctx.callSubagent`, or a resumed durable run started it. A subagent runs the hooks of the turn that started it, with the same `ctx.sessionId`, `ctx.caller` and `ctx.context`, so per-session hook state applies inside it. `preInput` does not run on a subagent's task, and `preOutput` does not run on a subagent's text, because that text goes to the agent or tool that called the subagent, not to a person. A subagent has no one to confirm, so `ask` denies the call. On a resumed durable run, `ctx.caller` is absent, and in-process state from the original turn is present only when the same process resumes it.

Hooks do not run on calls your own code makes: composite `ctx.callTool` children, `ctx.request()`, durable tool and intent calls, triggered tools, and direct tool evals. A subagent that such code delegates to does run hooks on its own calls. Enforce rules for author-made calls in the tool or the downstream API.

Folder shape

Hooks live at the **repo root** under `hooks/`, one folder per hook:

```
hooks/
└── ready-to-quote-guard/
    ├── hook.json     ← declarative manifest (binding, capabilities, config)
    └── index.mjs     ← decision module (exports createHook)
```

The manifest declares the binding; the module supplies only the decision function. Every subfolder of `hooks/` is loaded; there is no registration list.

Ship the handler as `index.mjs`. The loader imports the module without bundling it, unlike tools. `index.js` and `index.ts` are also probed, but a `.ts` handler works only where the host process can import TypeScript.

`hook.json`

```
{
  "name": "ready-to-quote-guard",
  "points": ["preToolUse"],
  "capabilities": ["reads_tool_io", "gates_tools"],
  "failPolicy": "closed",
  "priority": 10,
  "config": {
    "guardedTools": ["store__submissions__set"],
    "blockedRecommendation": "ready-to-quote"
  }
}
```

Field	Required	Meaning
`name`	no	Stable identifier, unique within the agent's hook chain. Defaults to the folder name.
`points`	yes	Lifecycle points to bind to (non-empty array; see below).
`capabilities`	yes	Array of capabilities. May be empty.
`failPolicy`	no	`"closed"` (a failing hook blocks) or `"open"` (a failing hook allows). Defaults per point: `closed` everywhere except `postTurn` (`open`).
`priority`	no	Number; lower runs first at a shared point. Default `0`.
`config`	no	Author-supplied defaults, passed to `createHook`. Overridable per agent via `amodal.json` (see Chain configuration).
`trustTier`	no	`"first_party"`, `"verified"`, or `"community"`. Metadata only; not enforced.

At runtime, a malformed hook (bad JSON, unknown point or capability, missing handler) is skipped with a `hook_skipped` warning, and the other hooks still load. A hosted deploy fails instead, naming each local `hooks/<name>/` folder whose `hook.json` is invalid or that has no handler module. The deploy check does not inspect the module's exports or hooks from installed packages. Two hooks with the same name keep the first one loaded, local before package, and log `hook_name_conflict`.

Hook points

Point	Fires	Payload
`preInput`	before a user/trigger message enters context (and before it is persisted)	`{ text, source }` where `source` is `'chat' | 'webhook' | 'automation' | 'channel'`
`preToolUse`	before a tool call executes, ahead of the permission gate	`{ toolName, args, connection?, endpointPath?, method?, intent?, isDelegated?, planModeActive? }`

Point	Fires	Payload
`postToolUse`	after a tool returns, before its result re-enters context	`{ toolName, args, status, result, durationMs }` (`result` is the tool's string output; structured results such as image blocks bypass this point)
`preOutput`	before assistant text is streamed, persisted, or routed to a channel; runs on every turn's text, not just the final answer	`{ text, channels }`
`postTurn`	after a turn completes; observe-only	`{ turn, usage, toolCalls, terminal }`

`preToolUse` runs before the connection policy (`policy.json`). The policy then evaluates the possibly rewritten args and has the final say.

Live root turns pass `source: 'chat'` to `preInput` and `channels: ['chat']` to `preOutput`. `isDelegated` and `planModeActive` are not populated in any loop. Do not base a permission decision on these absent fields.

When any hook is loaded, at any point, the root loop's assistant text is buffered instead of streamed token by token. The runtime runs the `preOutput` chain on each turn's text and emits the result as one block.

`postTurn` is accepted by the loader and test endpoint, but the live loop does not invoke it.

Handler contract

The module exports `createHook(config)` (or a default export) returning the decision function:

```
// hooks/ready-to-quote-guard/index.mjs

/** @param {Record<string, unknown>} config  merged hook.json config + amodal.json overrides */
export function createHook(config) {
  const guarded = new Set(config.guardedTools ?? []);

  return {
    /**
     * @param {string} point    the lifecycle point that fired
     * @param {object} payload  that point's payload (see table above)
     * @param {object} ctx      HookContext
     * @returns {Promise<{action: 'allow'} |
     *                   {action: 'block', reason: string} |
     *                   {action: 'modify', payload: object, note?: string} |
     *                   {action: 'ask', reason: string}>}
     */
    async run(point, payload, ctx) {
      if (point !== 'preToolUse' || !guarded.has(payload.toolName)) {
        return { action: 'allow' };
      }
      const row = payload.args?.value;
      if (row?.recommendation === config.blockedRecommendation) {
        ctx.log(`blocked recommendation in ${payload.toolName}`);
        return { action: 'block', reason: 'This workflow cannot set the ready-to-quote recommendation.'
};
      }
      return { action: 'allow' };
    },
  };
}
```

Accepted export shapes: a factory `(config) => {run}` or `(config) => run`, a plain `{run}` object, or a bare `run` function. `run` may be sync or async. A hook bound to several points receives each one through the same `run` and switches on `point`.

`ctx` (HookContext)

Property	Meaning
`config`	Reserved; always `{}`. The resolved config (hook.json `config` plus `amodal.json` overrides) is passed to `createHook(config)`; capture it in the factory closure rather than reading `ctx.config`.
`env(name)`	Read a runtime process environment variable. The hook runner does not apply an allowlist or require the `secrets` capability for this method.

Property	Meaning
`log(message)`	Structured log line, scoped to the session (`guardrail_hook` in runtime logs).
`signal`	`AbortSignal` ; aborts when the session or request is cancelled.
`agentId` , `scopeId` , `sessionId`	Identifiers for the current turn, when available.
`caller`	The verified caller of the turn: `{ userId?, source, orgId? }` , sourced from the authenticated request (JWT claims), never from the client-supplied `scopeContext` . Absent for unauthenticated triggers. Use this for permission decisions.
`context`	The turn's request context: string values the client sends with each turn, with token claims laid over them. Only keys pinned by a runtime token are verified, so use `caller` for permission decisions. Present when the turn carries context.
`store`	Read-only access to the agent's own stores: `get(store, key)` and `query(store, filter?)` . Present only when the hook declares the `store:read` capability. Reads are scoped to the agent and the caller's scope.

For a permission hook, check `ctx.caller.userId` against your own membership store with `ctx.store.get` , block when no row matches, and set `failPolicy: "closed"` so a failed read blocks.

Decisions

`run` returns one of four decisions:

Action	Effect
`{ action: 'allow' }`	No opinion; the chain continues.
`{ action: 'block', reason }`	Refuse. Terminal: short-circuits the rest of the chain.
`{ action: 'modify', payload }`	Replace the payload; the modified payload feeds the next hook, so transformers (successive redactors) compose. Valid at every point except `postTurn` .
`{ action: 'ask', reason }`	Require human confirmation. Non-terminal: a later hook may still block. Meaningful at `preToolUse` , where it routes the call through the standard confirmation flow.

When several hooks fire at one point, they run in ascending `priority` order and their decisions combine with precedence **block > ask > modify > allow**.

What `block` concretely does at each point:

Point	Effect of `block`
`preInput`	The turn is rejected before the message enters context or is persisted; the client receives an error event with your `reason` .
`preToolUse`	The tool never executes; the model sees an error tool result: `Blocked by guardrail: <reason>` .

Point	Effect of `block`
`postToolUse`	The result is masked: the model sees `[output withheld by guardrail: <reason>]`.
`preOutput`	The assistant text is replaced with `[response withheld by guardrail: <reason>]`.
`postTurn`	Ignored (observe-only).

A `preInput` modify rewrites the text that context, persistence, and the model see, and the client receives an `input_redacted` event. A `preOutput` modify is persisted to history, so the next turn sees what the user saw. A `preToolUse` modify rewrites `args` before the policy check.

Failure behavior

Each hook invocation is capped by a wall-clock timeout (default **1500 ms**). A hook that throws, rejects, times out, or returns no valid decision (`undefined`, or an `action` other than `allow`, `block`, `ask`, or `modify`) is converted to a decision by its failure policy:

- `failPolicy: "closed"`: the failure becomes a `block` (reason: `Hook "<name>" failed (fail-closed)`). This is the default at `preInput`, `preToolUse`, `postToolUse`, and `preOutput`.
- `failPolicy: "open"`: the failure becomes an `allow`. Default at `postTurn`; opt into it elsewhere only for pure observability hooks.

Every non-`allow` decision and every failure is logged (`guardrail_decision` in runtime logs) with the hook name, point, action, and reason.

Capabilities

Valid values are `reads_input`, `modifies_input`, `reads_output`, `modifies_output`, `reads_tool_io`, `modifies_tool_io`, `gates_tools`, `network`, `secrets`, and `store:read`. An unknown value skips the hook.

Only `store:read` is enforced: without it, `ctx.store` is absent. The others describe intent and do not restrict the module. `secrets` does not gate `ctx.env`.

Chain configuration (`amodal.json`)

The agent controls the composed chain through the optional `hooks` block:

```
{
  "hooks": {
    "order": ["pii-redactor", "ready-to-quote-guard"],
    "disabled": ["verbose-audit"],
    "config": {
      "ready-to-quote-guard": { "blockedRecommendation": "ready-to-quote" }
    }
  }
}
```

Key	Meaning
`order`	Explicit chain order by hook name. Listed hooks run first in this order; unlisted discovered hooks append after, sorted by their own `priority`.
`disabled`	Hook names to skip entirely.
`config`	Per-hook config overrides, keyed by hook name; merged over the hook's own `hook.json` `config`.

Hooks from packages

An installed package (declared in `amodal.json` `packages`) can contribute hooks: a hook package is a package with a `hooks/` directory inside it, scanned by the same loader. Select a subset with the `use` selector:

```
{
  "packages": [{ "package": "@acme/compliance-pack", "use": ["hooks.pii-redactor"] }]
}
```

A bare string entry loads everything the package ships (hooks included). A non-empty `use` list that names no `hooks.<name>` entries opts the package out of contributing hooks. An empty `use` list loads every hook. On a name conflict, the local `hooks/` folder wins. A declared but uninstalled package logs a warning rather than failing the agent.

Testing a hook

Run a hook against a fixture payload without driving a turn:

```
# inventory the hook manifests discovered under hooks/ and installed packages
curl $RUNTIME/inspect/hooks \
  -H "Authorization: Bearer $RUNTIME_TOKEN"

# run one hook (or the whole chain, omit hookName) at a point
curl -X POST $RUNTIME/inspect/hooks/test \
  -H "Authorization: Bearer $RUNTIME_TOKEN" \
  -H 'content-type: application/json' \
  -d '{
    "point": "preToolUse",
    "hookName": "ready-to-quote-guard",
    "scopeId": "taxpayer:100-00-0001",
    "context": { "taxpayer_id": "TP-1001" },
    "payload": {
      "toolName": "store__submissions__set",
      "args": { "value": { "recommendation": "ready-to-quote" } }
    }
  }'
```

The test context has no scope unless the request passes `scopeId`, so a hook that compares tool arguments against `ctx.scopeId` needs one to be tested. Likewise `ctx.context` is absent unless the request passes `context`, an object of string values that stands in for the turn's request context. A guard fixture sets it with a `context` field next to `scopeId`.

Both routes require an API key or an operator token; an end-user token receives `403`. The response reports the combined decision (`action`, `reason`), the input and output payloads, which hooks ran, and any `ctx.log` lines.

In Studio, the **Guardrails** page lists each hook and runs any `checks` fixtures declared in `hook.json` against the selected deployment's `/inspect/hooks/test` endpoint. Studio disables **Run** when the selected deployment was not built from the currently viewed source commit.

Local and packaged hooks run in the runtime process without a sandbox. Install only hook code you trust.

AGENT REPO

MCP Servers

Amodal connects to Model Context Protocol servers and exposes their tools to agents.

Configuration

Define an MCP connection in `amodal/connections/<name>/spec.json`:

```
{
  "protocol": "mcp",
  "transport": "http",
  "url": "https://tools.example.com/mcp",
  "auth": {
    "type": "bearer",
    "token": "env:TOOLS_TOKEN"
  }
}
```

Set `TOOLS_TOKEN` to the token alone; bearer authentication adds the `Bearer` prefix. This is the format generated by Studio's static bearer option and `amodal connections add tools --mcp <url> --auth bearer`.

An explicit header such as `"headers": {"Authorization": "env:TOOLS_AUTHORIZATION"}` also works. In that form, set `TOOLS_AUTHORIZATION` to the complete header value, including `Bearer` if required. Header references replace the whole value without adding a prefix.

For OAuth over `http` or `sse`, declare `"auth": {"type": "oauth2"}` and follow OAuth connections. Studio and CLI MCP creation default to OAuth, and Studio ships presets for common remote servers.

MCP connections need no `baseUrl`, `format`, or `policy.json`. The model receives the server's own tool descriptions and schemas. Connection policies, field restrictions, and `surface.md` do not apply to MCP calls.

Servers can also be defined under `mcp.servers` in amodal.json with the same `command`, `args`, `env`, `url`, and `headers` fields; `transport` is required there. A connection with the same name takes precedence.

Transports

Transport	Fields	Requirement
`stdio`	`command`, `args`, `env`	The default when `transport` is omitted. The runtime environment must contain the executable and its dependencies.
`sse`	`url`, optional `headers`	An accessible MCP server using SSE transport.
`http`	`url`, optional `headers`	An accessible MCP server using Streamable HTTP.

Credential-like `env` and `headers` entries, such as names ending in `token`, `key`, `secret`, or `password`, must be `env:NAME` references. A `scope:KEY` reference passes validation but is not resolved for MCP servers, so the server receives the literal string:

```
{
  "protocol": "mcp",
  "transport": "stdio",
  "command": "node",
  "args": ["/opt/tools/server.mjs"],
  "env": { "SERVICE_TOKEN": "env:SERVICE_TOKEN" }
}
```

Tool Discovery

Tools are named `<server>__<tool>`, using the connection directory name or the `mcp.servers` key:

```
internal-tools__lookup_ticket
internal-tools__add_comment
```

A named root agent must declare the server in its `mcp` list:

```
{
  "mcp": ["internal-tools"]
}
```

Use `mcp` for MCP servers, including servers defined as connections. Use `connections` for REST connections. Unscoped chat exposes all discovered MCP tools. If `agents/default/` exists, a chat request that omits `agent` uses that agent's `mcp` declarations. A delegated specialist declares individual inherited MCP tool names in `tools`.

Tools are discovered when the server connects. A server that fails to connect is logged and skipped, and the agent starts without its tools. **Test connection** in Studio lists a connected server's tools.

AGENT REPO

Custom Runtime Apps

A deployed agent serves Amodal's shared chat shell unless its repository ships its own single-page app. Enable a custom app in `amodal.json`:

```
{
  "name": "investor-relations",
  "version": "0.1.0",
  "runtimeApp": {
    "custom": true,
    "build": "npm run build",
    "dist": "dist"
  }
}
```

Field	Default	Description
`custom`	required	`true` builds and serves the repository's app. `false` uses the shared shell.
`build`	`npm run build`	Build command, run from the app folder after dependencies install.
`dist`	`dist`	Build output directory that Amodal serves.

The app lives in the folder holding `amodal.json`, which is the repository root unless deploy settings set a root directory. It sits beside `package.json` and the `amodal/` agent directories.

Build

Dependencies install with npm: `npm ci` when a `package-lock.json` is present, otherwise `npm install`. Commit a `package-lock.json` for pinned installs. A `pnpm-lock.yaml` is discarded, together with any `package-lock.json` beside it. The build command runs with a five-minute timeout and with `VITE_RUNTIME_URL` set to an empty string.

The deploy fails when the build command fails or the `dist` directory is missing. Read the build logs with `GET /api/builds/:id/logs`, fix the source, and redeploy. See Deployment API.

Amodal serves the deployed app from the agent's URL. Browser routes return `index.html`, and asset routes return the requested file.

Runtime URL

Make same-origin API calls. Derive the runtime URL from `window.location.origin` in browser code, and use `VITE_RUNTIME_URL` only for local development against a remote runtime. Requests such as `/chat/stream` and `/api/stores/investors` then reach the agent's own origin.

A runtime URL that is `undefined` when a request is built produces `undefined/chat/stream`, which the agent origin answers with the app's HTML. `@amodalai/react` rejects that response, and any stream response that is not `text/event-stream`, with an `SSEStreamError` naming the URL, status, and content type.

Minimal Wiring

```
npm install @amodalai/react react react-dom
```

```
// src/App.tsx
import '@amodalai/react/style.css';
import { AmodalProvider, AmodalChat } from '@amodalai/react';

const runtimeUrl = window.location.origin;

export function App() {
  return (
    <AmodalProvider runtimeUrl={runtimeUrl}>
      <AmodalChat />
    </AmodalProvider>
  );
}
```

`AmodalChat` uses the provider's `runtimeUrl` but not its `getToken`. When you authenticate with `getToken`, pass it to both components.

Chat requires runtime authentication even when the app is public:

- **Hosted team login:** make same-origin calls and omit `getToken`. The edge authenticates requests with the session cookie.
- **Product login:** point the SDK at an application proxy that holds runtime credentials and authorizes each scope, session, and operation. Return your application token from `getToken`.
- **Public app:** keep persistent keys on a backend. Public app access does not authorize chat or mint a token.

See Authentication, Embedding, and the React SDK.

Default chat uses the deployed agent definition. To limit its skills, connections, stores, and tools, declare `agents/default/`. See Agents.

AGENT REPO

Evals

An eval runs an agent scenario and checks the result. Define cases in `evals/`. Deterministic assertions inspect replies, tool events, and run metrics. Prose assertions are graded by an LLM judge or TypeSafe Jev.

Layout

```
evals/
├── refund-over-cap/
│   ├── eval.json
│   └── eval.md
├── ingest-report/
│   ├── eval.json
│   ├── eval.ts
│   └── report.pdf
├── triage.md
└── README.md
```

A directory case uses `eval.md` or `eval.ts`, with an optional `eval.json` manifest and fixtures. A flat Markdown file works for a case without supporting files. `README.md` is skipped.

Discovery is recursive. The ID is the path relative to `evals/`, without the file extension or trailing `/eval`: `refund-over-cap` or `journeys/onboarding/first-run`. Moving a case changes its ID, which identifies baselines and trends.

Top-level `evals/traces/` is reserved for online trace evaluators and excluded from eval discovery.

`eval.json`: metadata only

```
{
  "tags": ["refusal", "write-path"],
  "objectives": { "turns": { "max": 4 }, "hitlRequests": { "min": 1 } },
  "runs": 5,
  "passRate": 0.8,
  "timeoutMs": 60000
}
```

Field	Meaning
`tags`	Labels for `amodal eval --tag`.
`objectives`	Soft run-metric bounds; see Objectives.
`attachments`	Fixture files; see Attachments.

Field	Meaning
`runs`	Number of repetitions; default `1`.
`passRate`	Fraction of declared runs that must pass; default `1`.
`timeoutMs`	Deadline for each attempt in milliseconds, also bounded by the suite deadline.
`judgeModel`	Judge model as `provider/model`. Without it, the run's judge model applies (`ci.evals.judgeModel` or the run request), then `EVAL_JUDGE_MODEL`, then the agent's own model.
`grader`	Prose grader; see Grading with TypeSafe Jev.
Session	`agent`, `scopeId`, `scope`, and `maxSessionTokens`; see The session an eval runs in.

The same fields can be written in YAML frontmatter. Declaring a field in both frontmatter and `eval.json` is a load error.

`eval.md`: the case

```
# Eval: Looks up the order before answering

Check that the answer comes from the order system.

## Setup

Context: The operator is viewing order ORD-1001.

## Query

"What's the status of ORD-1001?"

## Assertions

- tool_called_with: lookup_order {"order_id": "ORD-1001"}
- contains: delivered
- no_failed_actions
- Should state the order status clearly.
- Should NOT invent a delivery date the tool did not return.
```

Section	Meaning
`# Eval: …`	Title. Text before the first `##` section becomes the description.
`## Setup`	A `Context:` line, prepended to the first user message. `App:` and `Runs:` lines are accepted but do not affect the run. Other lines are a load error.
`## Query`	One user message. `## Conversation` replaces it when both exist.

Section	Meaning
`## Conversation`	A script of messages and actions.
`## Assertions`	One bullet per check. A case without assertions is a load error.

Set repetitions with `runs` in metadata, not a `Runs:` line. Set `scopeId` and `agent` in metadata, not in `## Setup`. A `## Objectives` section is a load error.

Grading with TypeSafe Jev

Prose assertions use an LLM judge unless the case or the run selects TypeSafe Jev. Save grading settings in `eval.json` or frontmatter:

```
{
  "grader": {
    "kind": "typesafe",
    "model": "jev-latest",
    "passAbove": 0.7,
    "fallback": {
      "model": "anthropic/claude-haiku-4-5-20251001",
      "uncertain": { "low": 0.3, "high": 0.7 }
    }
  }
}
```

`kind` is `typesafe` or `llm_judge`; both accept an optional `model`. `passAbove` passes each prose assertion at or above a probability. For scored grading, use `rubric: { "levels": ["Not satisfied", "Partly satisfied", "Fully satisfied"], "passAt": 2 }` instead. Levels are numbered from zero, and a rubric accepts 2 to 10 nonempty levels.

The optional `fallback` sends uncertain answers to an LLM. Statement probabilities within `uncertain`, including both endpoints, use the fallback. A rubric's fallback uses `confidenceBelow`, such as `0.6`, instead of `uncertain`. A failed grading call is reported as **Not graded**.

Deterministic checks, named scorers, and code-authored evals keep their own rules. In Studio, **Run settings** on the **Evals** page can override the saved grader for a run; **Saved with each case** uses the saved grader.

Standalone runtimes require `TYPESAFE_API_KEY`; hosted runs use the platform's TypeSafe configuration. Each Jev request has a 30-second ceiling. Responses over 100,000 characters require an LLM judge.

Repeating an eval

With `"runs": 5, "passRate": 0.8`, at least four of the five runs must pass. The required count is `ceil(runs × passRate)`. Execution stops when the verdict is settled; here a second failure makes a pass impossible, so the remaining attempts are skipped.

Each attempt gets its own `timeoutMs`, bounded by the time left in the suite's `--timeout`. A deadline that stops the repeats does not reduce the number needed to pass. The result reports attempted, passed, required, and total runs, and includes the first failing attempt when the case fails.

The session an eval runs in

An eval without `agent` uses the chat route's default selection: `agents/default/` if present, otherwise unscoped chat.

```
{
  "agent": "request-intake",
  "scopeId": "user-a",
  "maxSessionTokens": 4000,
  "scope": { "mission_id": "m-42" }
}
```

Field	Meaning
`agent`	Named agent under `agents/`. An unknown name is an error.
`scopeId`	Data scope identifier for tools, memory, and non-shared runtime stores.
`scope`	Per-turn context, equivalent to chat request `context`, used by `CONTEXT.md`.
`maxSessionTokens`	Session token ceiling; exhaustion ends with `budget_exceeded`.

Each attempt creates a fresh session. Setting `scopeId` does not supply authenticated user claims.

The answer model is selected for the whole suite: `amodal eval --model` sets it when the CLI starts the runtime, and a `--url` target uses its own model. A case cannot pin an answer model or force a skill. Use `skill_loaded:` to check whether the model chose the expected skill.

Assertions

Deterministic keys

Key	Checks
`contains: <text>`	Reply contains text.
`regex: /pattern/flags`	Reply matches a regex. Bare patterns also work.
`starts_with: <text>`	Reply begins with text.
`length_between: [min, max]`	Reply length in characters.
`tool_called: <name>`	Tool executed.
`tool_not_called: <name>`	Deprecated alias of `not tool_called: <name>`.

Key	Checks
`tool_called_with: <name> {json}`	A call's arguments match a JSON subset.
`tool_returned: <name> {json}`	A call's result matches a JSON subset. Non-object returns are wrapped under `result`.
`tool_result_contains: <name> <text>`	A result contains a substring.
`tool_succeeded: <name>`	Tool executed without an error.
`tool_order: a, b, c`	Tools ran in that relative order.
`subagent_called: <name>`	Specialist was invoked.
`skill_loaded: <name>`	Skill was activated.
`hitl_requested`	Run requested human input of any kind.
`hitl_requested: <subject>`	A confirmation for that endpoint, or a durable approval for that tool, was requested.
`no_failed_actions`	No failed tool or subagent, error, or warning.
`max_turns: <n>`	Maximum model round trips.
`max_hitl: <n>` / `max_compactions: <n>`	Maximum human-input requests or compactions.
`max_latency: <ms>`	Maximum wall-clock duration.
`scorer: <name>`	Run a named scorer.

An unknown key shaped like `snake_case:` is a load error. Other text is judged prose.

Prefix a key with `not` or `Should NOT` to invert it. Use an argument-level negative when the tool may run for other records:

```
- tool_called_with: update_request {"request_id": "R-16"}
- not tool_called_with: update_request {"request_id": "R-11"}
```

Pair `tool_succeeded` with `tool_returned` when the value matters. A JSON subset such as `tool_returned: combine {"sum": 14}` distinguishes `14` from `1400`; a substring check does not.

Judged prose

Bullet	Effect
`Should …`	Required behavior.
`Should NOT …`	Required absence.

Bullet	Effect
`May …`	Optional behavior. Graded and reported, but excluded from pass/fail.

Conversation

```
## Conversation

- user: "Refund ORD-1002 in full, $940."
- deny: issue_refund
- user: "What would it take to approve it?"
```

Step	Meaning
`user:`	Send a message.
`approve:` / `deny:`	Supply the turn's confirmation decision.
`choose:`	Answer `ask_choice` by option label.
`tool:`	Invoke a tool directly without model selection.

Confirmations default to denied. The first `approve:` or `deny:` after a user message controls all of that turn's confirmation requests. Its tool name is a label, not a selector. Use separate user turns for actions needing different decisions, and assert the actual operation and arguments.

For a native OpenAPI write, denial happens before execution:

```
## Conversation

- user: "Record that Lee Park is adopting Biscuit."
- deny: shelter\_\_adoptResident

## Assertions

- hitl_requested: /residents/{residentId}/adopt
- not tool_succeeded: shelter\_\_adoptResident
- no_failed_actions
```

This assumes a `shelter` connection with confirmation required on the adoption operation. A confirmation question in ordinary prose does not exercise the runtime's approval gate.

Calling a tool directly

A `tool:` step tests a handler without relying on the model to choose it:

```
## Conversation

- tool: build_from_packet {"packet_id": "PK-1"}
- user: "What did the packet build?"

## Assertions

- subagent_called: package-reviewer
- tool_returned: build_from_packet {"status": "staged"}
```

Arguments must be a JSON object and can be omitted for a tool with no parameters. Missing required arguments fail validation before execution. Nested tool and subagent events are recorded for assertions. A script can contain only tool steps.

Direct invocation skips the outer permission checker, `preToolUse` and `postToolUse` hooks, and `requiresConfirmation`, so it can execute a write without asking. The handler's own checks still run. `ctx.request()` responses keep their field restrictions. A subagent the tool delegates to runs its model's calls through hooks and endpoint policy. Test approval with `user:` and `deny:`. An approval step directly after a `tool:` step is a load error.

Attachments

Declare fixture files in the manifest and attach them to user turns by name:

```
{
  "attachments": [{ "path": "report.pdf" }, { "name": "annex", "path": "fixtures/annex-b.pdf" }]
}
```

```
- user: "Summarize this report and its annex." [attach: report.pdf, annex]
```

Paths resolve against the eval directory. A name defaults to the filename. Only user turns can attach files.

MIME type is inferred for `.pdf`, `.png`, `.jpg`, `.jpeg`, `.gif`, `.webp`, `.txt`, `.md`, `.csv`, `.docx`, `.pptx`, `.xlsx`, `.odt`, `.odp`, `.ods`, and `.rtf`. Other extensions require `mimeType`. Office and OpenDocument files are converted to text at ingestion.

An undeclared attachment name fails loading. An unreadable file or an unknown extension without a MIME type fails the eval when it runs, before the first model call. Assert a fact found only inside the fixture to verify that the document reached the model.

Objectives

Objectives are soft bounds used for ranking, not pass/fail. Supported metrics are `turns`, `userTurns`, `toolCalls`, `redundantToolCalls`, `subagentCalls`, `hitlRequests`, `failedActions`, `replyChars`, `latencyMs`, and `costMicros`. Each takes a `min`, a `max`, or `"minimize"`. An unknown metric is a load error.

Use an assertion when a bound must fail the case: `max_turns: 4` is a gate, while an objective on `turns` is a ranking signal.

Code-authored evals

Use `eval.ts` for branching conversations, custom predicates, or assertions scoped to one turn:

```
export const readsBothDocuments = {
  description: 'Read facts from documents as they arrive.',
  async test(t) {
    const first = await t.send('Read this report.', { attachments: ['report.pdf'] });
    t.require('report reached the model', () => /Project Cedar/.test(first.reply ?? ''));
    first.satisfies('does not invent the annex amount', () => !/1572/.test(first.reply ?? ''));

    const second = await t.send('Read its annex.', { attachments: ['annex.pdf'] });
    second.satisfies('reads the annex amount', () => /1572/.test(second.reply ?? ''));
    t.noFailedActions();
  },
};
```

Declare both attachments in the sibling manifest. One file can export several cases. A named export contributes to the case ID, such as `ingest/readsBothDocuments`. An exported object can set `tags`, `objectives`, `runs`, `passRate`, `timeoutMs`, `judgeModel`, `agent`, `scopeId`, and `maxSessionTokens`, which override the file's `eval.json`. Set `scope`, `attachments`, and `grader` in `eval.json`; `t.send` can also pass `scope` per turn.

The context `t` and returned turn objects expose `calledTool`, `notCalledTool`, `calledSubagent`, `loadedSkill`, `toolOrder`, `messageIncludes`, `maxTurns`, `maxToolCalls`, `noFailedActions`, `succeeded`, and `satisfies(label, predicate)`. Methods on a returned turn inspect only that turn. `calledTool(name, {input, output})` accepts a JSON subset or a predicate for each matcher.

Method	Purpose
`t.send(message, {approve, attachments, scope})`	Drive a conversation turn. `approve` defaults to `false`.
`t.callTool(name, args)`	Invoke a tool directly, with the same ungated behavior as a `tool:` step.
`t.simulateUser(opts)`	Let a model play the user. Requires `instructions`, `opening`, `maxUserTurns`, and `stopTokens`.
`t.require(label, predicate)`	Record a precondition; stop with failure if it fails.
`t.skip(reason)`	Mark the case not applicable and stop.
`t.log(message)`	Add a note to the result.

Ordinary assertions collect failures and continue.

Scorers

A scorer is a reusable check in `scorers/<name>/scorer.ts`. Reference it with `scorer: <name>`.

```
const known = new Set(['1', '2.1', '2.2']);

export default {
  description: 'Every cited section exists in the reference index.',
  requires: ['finalMessage'],
  preprocess: ({ facts }) => {
    const cited = [...(facts.finalMessage ?? '').matchAll(/§\s*([\d.]+)/g)].map((m) => m[1]);
    return { cited, missing: cited.filter((id) => !known.has(id)) };
  },
  score: ({ pre }) => (pre.cited.length === 0 ? 1 : 1 - pre.missing.length / pre.cited.length),
  reason: ({ pre }) =>
    pre.missing.length ? `Unknown sections: ${pre.missing.join(', ')}` : 'All cited sections exist',
};
```

A `scorer:` assertion passes only when the score is `1`. Scores are clamped to the range 0 to 1, and a scorer that throws scores `0`.

`requires` declares the facts a scorer reads: `finalMessage`, `toolCalls`, `toolArguments`, `subagentCalls`, `skillsActivated`, `hitlRequests`, or `turns`. A scorer that requires a fact the run cannot supply, or an unknown fact name, scores `0` and names the missing facts.

Running

```
amodal eval
amodal eval journeys/
amodal eval '/^(smoke|journeys)\//'
amodal eval journeys/ --tag safety --tag p0
amodal eval --json
amodal eval --url https://your-runtime.example.com
```

The positional filter matches an ID, directory prefix, substring, or slash-delimited regex. Multiple tags are OR'd; the tag filter and positional filter are AND'd.

Without `--url`, the CLI starts the runtime that `amodal dev` runs, on an ephemeral port. With `--url`, it runs the evals provided by that runtime, with its configured providers. See CLI for the other flags.

The command exits nonzero if any case fails or no cases run. When the suite exceeds `--timeout`, the report retains completed results and identifies the incomplete run.

Running in Amodal

On the agent's **Evals** page, run one case or **Run all**. Filtering the list turns **Run all** into **Run N shown**. Selecting several models (and optional per-model effort levels) under **Run settings** compares them on the same cases:

each case's comparison table shows the verdict, checks, attempts, duration, and combined answer + grading cost for every selected model, and badges the **Lowest cost passing result**.

Runs execute from the browser tab. Refreshing, closing the tab, signing out, or leaving the agent workspace aborts unfinished runs; navigating within the workspace does not. **Stop run** in the progress banner stops the active case and skips the queued ones. Completed results saved to the agent's history appear under **Run history** on the case.

Hosted eval sessions are not saved to conversation history. Agent and judge calls still produce traces and incur model usage charges.

Online trace evaluators

Markdown files under `evals/traces/*.md` define online evaluators that grade recorded runs on **Traces > Evaluators** and sync automatically on deploy (`repo` badge):

```
# Trace Eval: Grounded Tax Ruling Citations

Check that cited IRS sections match the retrieved knowledge.

## Config

Kind: llm_judge
Model: google/gemini-3.5-flash
Sampling: 100%
Enabled: true

## Filters

Source: deployed
Status: ok
Tool: load_knowledge

## Assertions

- Should cite only rulings and form numbers returned by tool calls in the trace.
```

Each file defines one evaluator (`## Assertions` must contain a single `-` bullet). `## Config` accepts `Kind` (`llm_judge` , the default, or `typesafe`), `Model` , `Sampling` (`0%` to `100%`), and `Enabled` (`true` or `false`). `## Filters` optionally narrows eligible runs by `Source` (`local` or `deployed`), `Status` (`ok` or `error`), `Model` , and `Tool` . For `Kind: typesafe` , add a `## Question` section (`Type: noul` with `Pass above:` or `Type: score` with `Pass at:` and a `## Levels` bullet list) and an optional `## Fallback` section (`Model:` plus `Uncertain: 0.3 to 0.7` or `Confidence below: 0.6`).

Deployment triggers

Deployment evals are opt-in. For GitHub-connected agents, set `ci.evals.runOn` in `amodal.json` to run the suite after a successful deployment:

```
{
  "ci": {
    "evals": {
      "runOn": "production"
    }
  }
}
```

`ci.evals.runOn`	Automatic execution
`"never"` or omitted	Skip deployment evals. This is the default.
`"production"`	Run on production deployments only.
`"all"`	Run on every deployment.

`ci.evals` also accepts `runs`, `passRate`, and `judgeModel` as suite defaults. A case that sets its own `runs` or `passRate` keeps both of its values, and a case's `judgeModel` overrides the suite's.

Cloud reads the configuration at the deployed commit. Missing or unreadable configuration, invalid JSON, and an unrecognized `runOn` value skip evals. A triggered run posts an `amodal/evals` commit status, which gates promotion; see Deployment API. Skipped evals post no status.

Amodal-hosted source has no deployment trigger; run its evals from the **Evals** page or with `amodal eval`.

AGENT REPO

Confirmation Model

Approval applies to calls the model selects in the agent loop. Durable child-tool execution, authored `ctx.request()` calls, and direct tool evals bypass these gates. Inline composites refuse confirmation-required and connection-category children. Enforce authorization and approval for those paths in the workflow or the downstream API. See Agent Workflows.

Tiers

Tier	Behavior
Auto-approve	The call runs without user interaction.
Confirm	The runtime emits `confirmation_required` and runs the call after approval.
Review	The runtime refuses the call. Any human-review workflow belongs to the host application.
Never	The runtime refuses the call.

Surface	Where configured	Values	Tier
Connection endpoint	`policy.json` `confirm`	omitted / `true` / `"review"` / `"never"`	auto / confirm / review / never
Custom tool	`tool.ts` `exposure.kind`	`open` / `requires-confirmation` / `operator-gated`	auto / confirm / not enforced
Request tool call	`intent` argument	`"read"` / `"write"` / `"confirmed_write"`	which side of the handshake a call is on

See policy.json for the endpoint format, matching rules, and threshold escalation.

Custom tools

- `open` (the default): the model calls the tool freely.
- `requires-confirmation`: every model-selected call pauses for approval. The card's reason is `Tool "<name>" requires user confirmation`.
- `operator-gated`: a label for tools run from operator UI. The runtime does not enforce it. Set `llm_callable: false` to keep the tool out of the model's tool list while composites and triggers can still call it.

In `tool.json`, `confirm: true` or `"review"` allows write-intent `ctx.request()` calls but does not prompt the user. `confirm: "never"` skips registration with a warning. See Tools.

The request tool handshake

Every `request` call declares `intent`. POST, PUT, PATCH, and DELETE must declare `"write"` or `"confirmed_write"`, unless the matching endpoint policy sets `"intent": "read"`.

When a write hits a confirm-tier endpoint:

1. The model calls with `intent: "write"`. The loop parks the call and emits `confirmation_required`.
2. The user approves. The SDK chat components render the card.
3. The runtime runs the same call with `intent` rewritten to `"confirmed_write"`. The model does not call again.

Outside the agent loop, such as in durable execution, a confirm-tier `"write"` returns a preview object that tells the caller to call again with `"confirmed_write"`.

Native operation tools

Native OpenAPI tools take no `intent`. Reads and writes follow the method and policy. Invalid arguments are refused before a confirmation card. The card carries the connection, method, path template, policy reason, and grouped arguments. Approval is bound to those effective arguments, so a hook rewrite after approval asks again.

The `confirmation_required` event

The SSE event (`SSEConfirmationRequiredEvent`) has `endpoint`, `method`, `reason`, `escalated`, and `timestamp`, plus optional `params`, `connection_name`, and `correlation_id`. It fires for:

- A connection write at the confirm tier.
- A `requires_confirmation` custom tool, or a file tool that writes, edits, or deletes a repo file.
- A hook `ask` decision.

See SSE event types for the client view and Interactions for the approval response.

Escalation

Thresholds and `delegations.escalateConfirm` raise a tier at call time. Both escalate only to `review` or `never`, so an escalated call is denied with an escalation reason and emits no `confirmation_required` event. Subagents have no approval handler, so a confirm-tier call from a subagent is denied either way.

AGENT REPO

Security & Guardrails

This page lists what each runtime control covers and where it stops. Configuration lives on the linked pages.

Authentication and scope

Hosted chat, session, and store requests require runtime authentication. Public app access does not grant chat access. See Authentication.

Authorize on verified identity. A verified JWT `scope_id` takes precedence over a client-supplied scope. Request context can carry untrusted values even when a JWT supplies other context keys.

Session history and resume enforce the scope a verified token is pinned to. A token without a pinned scope, such as an API key or operator token, is not limited to one scope. For a multi-tenant application, keep runtime credentials on your backend and authorize every scope, session, and operation before proxying it. A browser-held token without a pinned scope bypasses those checks. See Embedding.

Coverage

Control	Applies to	Does not apply to
Hooks	Model tool calls and string tool results in the root chat loop and subagent loops; input and assistant text in the root chat loop	Calls your own code makes (composite child calls, `ctx.request()`, durable intents, triggered tools, direct tool evals), structured tool results, and reasoning
Endpoint policy and confirmation	Model calls to `request` and native OpenAPI tools	`ctx.request()` in custom tools, MCP tools, and connection-driver tools
Field restrictions	Responses to `request`, native OpenAPI tools and `ctx.request()`, on every path	MCP tools, connection-driver tools, and the model's reply

Enforce permissions for uncovered paths in the tool handler or the downstream API. Use a `preOutput` hook for redaction of the model's reply.

Field restrictions apply only when the response shape matches the policy. Verify removal and redaction with an eval that contains the values to withhold.

Resource access

An agent's declared `tools`, `skills`, `connections`, `stores`, `mcp`, and `subagents` determine its session's resources. Each entry names one resource exactly, with no wildcard. A name that matches no loaded resource grants nothing. Conditional entries in `agent.ts` can check verified claims through `ctx.claims`. See Agents.

Runtime HTTP roles (`user`, `admin`, `ops`) are separate from these declarations.

Execution limits

Limit	Default	Effect
Model turns	50	Stops a chat run at the turn limit.
Turn deadline	10 minutes when the caller passes no signal	Aborts the turn through its cancellation signal.
Hook call	1500 ms	A timed-out hook follows its `failPolicy`.

PLATFORM

Amodal Configuration

Amodal has two configuration scopes:

- **Agent workspace** pages live under `/agents/:agentId/*` and configure one agent.
- **Organization workspace** pages live under `/org/:orgId/*` and configure members, billing, model provider credentials, and automation tokens.

Permissions decide what each member can see and edit. Viewers see read-only versions of pages that admins and developers can edit.

Agent Workspace

The sidebar lists **Overview** at the top, groups agent pages into **Agent**, **Deploy**, and **Operate**, and places **Settings** at the bottom.

Page	Path	What it shows
Overview	`/agents/:id`	Health summary, recent sessions, billable spend, model status, connection gaps, and suggested actions.
Models	`/agents/:id/models`	Model selection, billing-period usage, provider key status, and model comparison. See Models.
Connections	`/agents/:id/connections`	Connection inventory, credential status, runtime health, and live connection tests.
Guardrails	`/agents/:id/guardrails`	Hook inventory, available hook packages, and fixture test runner against the selected deployment.
Automations	`/agents/:id/automations`	Scheduled (`cron` with time zone, `every`, or one-shot `at`) and event-driven background bindings and their runs.
Stores	`/agents/:id/stores`	Store schemas and documents.
Memory	`/agents/:id/memory`	Memory entries when memory is enabled.
Prompt	`/agents/:id/prompt`	Compiled system prompt, token budget, and the largest prompt contributors.
Knowledge	`/agents/:id/knowledge`	Markdown knowledge documents, searchable corpora, and runtime libraries with their filed documents.
Source	`/agents/:id/files`	Source browser and editor for Amodal-hosted versions or a connected Git repository.
Proposals	`/agents/:id/proposals`	Source changes proposed from agent chat, the builder chat, or issue fixes. See Proposals.

Page	Path	What it shows
Deploys	`/agents/:id/deploys`	Deploys, build logs, deploy settings, and promote, start, stop, redeploy, and delete actions.
Evals	`/agents/:id/evals`	Eval cases and runs, including multi-model comparison. See Evals.
Embed	`/agents/:id/embed`	See Embed.
Sessions	`/agents/:id/sessions`	Saved conversations with scopes, token usage, messages, tool calls, and online evaluator review pass rates.
Traces	`/agents/:id/traces`	Recorded runs with labeled spans and waterfall detail, trace sessions, monitoring, and online evaluators.
Issues	`/agents/:id/issues`	Recurring failures that triage found in flagged production runs. See Issues.
Feedback	`/agents/:id/feedback`	End-user thumbs-up and thumbs-down ratings and comments linked to sessions and runs.
Logs	`/agents/:id/logs`	Full-width logs explorer with a **Runtime** / **Audit** toggle, volume histogram, live tail, and facets.
Activity	`/agents/:id/activity`	The **Audit** view of **Logs**: deploys, source commits, proposal decisions, auth changes, and secret edits.
Cost	`/agents/:id/cost`	Estimated model spend by scope, model, deploy, and day.
Settings	`/agents/:id/settings/general`	See Settings.

The **Sources** switcher in the sidebar footer selects which commit or branch Studio views (`viewing`, `current`, and `deployed` badges) so you can inspect or roll back earlier agent definitions. Selecting a deploy switches the viewed source to the commit that deploy was built from. On **Guardrails**, running a hook fixture targets the selected deploy's runtime and is disabled with a warning when that deploy was not built from the viewed commit.

On **Sessions**, each conversation shows its aggregated pass rate from online trace evaluators. Opening a session shows a **Session review** card where **Run review** (**Review N unjudged runs**) grades past turns in that session that have not yet been evaluated.

Traces > Sessions can list sessions that have no saved conversation. These rows show **Conversation unavailable**, and their runs remain available.

Under **Traces > Evaluators**, online evaluators grade recorded runs. Choose an **LLM judge** with a rubric, or **TypeSafe Jev** with a statement or scoring rubric. TypeSafe Jev can escalate uncertain results to an LLM judge. Set a sampling percentage and filter eligible runs by source, status, model, or tool. Evaluators committed in the repository under `evals/traces/*.md` sync on deploy and appear with a `repo` badge; see Online trace evaluators.

Logs opens a full-width explorer shared by **Runtime** (`/agents/:id/logs`) and **Audit** (`/agents/:id/activity`). **Runtime** draws a stacked volume histogram by level, supports 5-second **Live** polling on page 1, and filters by

time range (`24h` , `7d` , `30d` , `90d` , `all`), search (`/` , including `key:value` attribute queries), multi-select level (with a **Problems** shortcut for `warn` + `error`), context (`chat` , `tool` , `guardrail` , `oauth` , `mcp` , `knowledge`), and the top 20 scopes. **Audit** filters by severity, source, and deploy. Clicking any row opens a detail drawer with one-click filter chips, flattened attributes, raw JSON, and `j` / `k` keyboard navigation.

Cost estimates model spend from trace spans. It excludes platform fees and reports runs with missing pricing. The **Billable spend** chart on Overview and **Organization > Usage** show billing ledger charges, including platform fees and provider pass-through.

Settings

Tab	Path	Purpose
General	`/agents/:id/settings/general`	Agent name, agent ID, app URL, and creation date.
Source	`/agents/:id/settings/source`	Source provider, current ref, and GitHub export or connection.
Secrets	`/agents/:id/secrets`	Agent environment variables and connection secrets.
API Keys	`/agents/:id/settings/api-keys`	`ak_` runtime keys.
Auth	`/agents/:id/auth`	Hosted app access and end-user identity.
Runtime	`/agents/:id/runtime`	Runtime machine memory.
Domains	`/agents/:id/settings/domains`	Approved embed origins and custom app hostnames.

The General display name is platform metadata. It is separate from `amodal.json#name` .

Source

An agent's source is either Amodal-hosted versions or a connected GitHub or GitLab repository. For a GitHub source, this tab can connect, reconnect, disconnect, or choose from the repositories granted to the Amodal GitHub App.

Export to GitHub moves Amodal-hosted source to GitHub. It requires an empty repository that the Amodal GitHub App can access. Export commits every saved version to that repository and switches the agent's source to GitHub.

Push deployment rules and deployment evals live in `amodal.json` , not in this tab. See CI Rules.

Secrets

Secrets are agent environment variables. Source files reference them with `env:NAME` . Some platform-managed secrets cannot be edited or deleted.

An `admin` automation token can list secret names and create, update, or delete secrets. No automation token can read a secret value.

API Keys

Agent API keys are persistent `ak_` credentials for calling the deployed runtime from a server. The raw key is shown once at creation. See Authentication.

Auth

See Authentication for app access modes, runtime credentials, and identity settings.

Runtime

Memory	Use
512 MB	Default. Fits most agents.
1 GB	Larger bundles and several connection drivers.
2 GB	Heavy in-process workloads.
4 GB	Maximum. Use when smaller sizes run out of memory.

A change applies on the next deploy or runtime rollout.

Domains

Section	Purpose
Approved Domains	Hostnames, `localhost`, or wildcards such as `*.example.com`. They do not route app traffic.
Custom Domains	Concrete hostnames that serve the hosted agent app. `localhost` and wildcards are not accepted.

The edge does not restrict runtime calls to approved domains. It accepts cross-origin requests from any site and authenticates them by bearer token.

Custom domain status is `pending`, `verifying`, `active`, or `error`.

Embed

The Embed page configures the preview and the generated `@amodalai/react/widget` snippet.

Field	Options or meaning
Enabled	Turns the widget on or off in the saved config.
Position	`floating`, `right`, `bottom`, or `inline`.
Scope	`optional` or `required`. `required` makes `scopeId` a required prop of the generated component.
Open by default	Opens the widget panel on load.
Session history	Shows previous sessions.
Feedback buttons	Shows thumbs up and down controls.
Verbose tools	Shows expanded tool details.
Header	Widget title text.

Field	Options or meaning
Placeholder	Input placeholder.
Empty state	Text shown before the first message.
Primary color	Accent color.
Mode	`auto`, `light`, or `dark`.
Allowed domains	One origin per line, saved with the config. The runtime does not enforce this list.

A mounted widget uses its own props. Saving Embed settings does not change widgets already installed in another app. See Chat for the widget props.

Models

The Models page selects the agent's model from the catalog. It also sets an effort level for models that support one, a thinking budget, an output-token limit, and the models bound to the `simple` and `advanced` tiers. An unbound tier uses the agent's model. See Agents.

Below the configuration card, the page shows the agent's billing-period usage (cost, tokens, and sessions), **Provider keys** verification status (`Verified`, `Key issue`, or `Missing`), and a searchable, sortable **Compare models** table (input and output price per 1M tokens, context window, current billing-period usage, and availability status) with a **Use** action on each row.

Organization-level provider credentials, custom base URLs for private or on-prem LLMs, and model allowlists live in **Organization > Model Providers**.

Organization Workspace

Page	Path	Purpose
Agents	`/org/:orgId`	List and create agents.
Alerts	`/org/:orgId/alerts`	Failure alerts across agents, such as failed runs, model and tool errors, and failed deploys.
Users	`/org/:orgId/members`	Invite users, manage roles, revoke invites, and remove members.
Usage	`/org/:orgId/usage`	Usage charges by organization and agent.
Billing	`/org/:orgId/billing`	Plan, credits, spend limit, payment methods, and subscription actions.
Model Providers	`/org/:orgId/model-providers`	Provider enablement, Amodal-managed or your own API key, base URL, verification, and allowed models.
Marketplace	`/org/:orgId/marketplace`	Publish, edit, and archive your template and plugin listings.
Settings	`/org/:orgId/settings`	Organization name, ID, plan, and creation date.

Page	Path	Purpose
GitHub Access	`/org/:orgId/github-access`	Install the GitHub App and manage its installations. Shown to admins of company organizations.
Automation Tokens	`/org/:orgId/automation-tokens`	`pk_` tokens for coding agents and CI.

Roles are assigned per organization. Admins and developers are billable seats; viewers are free. A pending invite is billed only after acceptance. You cannot remove yourself, and the last admin cannot be demoted or removed.

Verify a model provider's credentials before sending production traffic.

Automation Tokens

Automation tokens are `pk_` credentials for the Platform API. They cannot call runtime chat; exchange one for a runtime token first. See Authentication.

Kind	Access
`observer`	Read-only status, deploys, sessions, usage, and evals.
`operator`	Observer access plus redeploy, deployment, eval, and short-lived runtime token minting (`POST /api/agents/:id/tokens`). No secrets, billing, or members.
`admin`	Administration, including creating agents, connecting GitHub source, and managing secrets.

A token is bound to one agent or to every agent in the organization. The UI offers 7, 30, or 90 days of validity; the API accepts `expires_in_days` from 1 to 365. Admin tokens default to 7 days and other kinds to 30. The raw token is shown once at creation and once after each rotation.

Every token operation requires a signed-in user session, so automation tokens cannot create, rotate, or revoke tokens. Listing, rotation, revocation, org-wide tokens, and admin tokens require organization admin permissions. An agent-bound `observer` or `operator` token requires permission to edit that agent plus the organization permissions its kind grants.

Use an org-wide admin token only for setup that creates agents or connects source. Use agent-bound `operator` tokens for CI deploys and `observer` tokens for monitoring.

PLATFORM

Issues

An issue is a recurring failure in a deployed agent: a named pattern, the runs that show it, and whether the last change fixed it. Review issues on the **Issues** page in Studio. Runs from `amodal dev` are not triaged.

Where issues come from

A run is flagged when any of these is true:

Signal	Recorded by
Evaluator fail	an online trace evaluator under **Traces → Evaluators** failed the run
Thumbs down	an end user rated a reply down in the chat widget
Annotated	an operator marked the run thumbs down in the trace viewer

Every fifteen minutes the platform triages each flagged run once. The issue model writes a one-line diagnosis and assigns the run to a known issue or opens one. **Scan now** runs the triage for one agent immediately. Triage model calls are billed to the organization like online evaluators, and stop at the organization's billing limit.

Settings

The settings on the Issues page apply to one agent.

- **Triage flagged runs.** On by default. Turn it off to stop new issues and the model calls behind them.
- **Model for issue triage and draft fixes.** The agent's own model by default. Another model here does not change the model the agent runs on.

Lifecycle

Status	Meaning
Open	Runs keep matching the issue.
Fix shipped	A deploy is linked as the fix. Runs are still matched against the issue.
Resolved	Someone reviewed the fix and marked the issue resolved.
Regressed	A run on the fix deploy or a later production deploy matched again.
Dismissed	Someone decided the issue is not a problem. Matching runs are absorbed without a new issue.
Merged	Someone merged it into another issue, which received its runs.

Resolution is manual. Runs from older deploys or previews do not regress a linked fix.

The **Issues** page groups issues under **Active** (`Open` , `Fix shipped` , and `Regressed`), **Resolved**, and **Dismissed** tabs. Opening an issue shows its 30-day matched-runs chart, the flagged runs and their triage diagnoses, and

management actions: **Edit issue** (updates the title and matching description), **Dismiss** (requires a reason), **Merge into...** (moves the issue's runs into another active issue), and **Reopen** (for resolved or dismissed issues).

Fixing an issue

The **Fix** section of an active issue shows three steps and offers the action for the current one:

1. **Draft. Draft a fix** asks the issue model to propose edits to `AGENT.md` and the `.md` files under `agents/`, `skills/`, and `knowledge/`. Drafting runs in the background. The draft reads the files where approving checks them: the production branch of a GitHub source, or the latest version of an Amodal-hosted source.
2. **Review and approve.** The draft shows its reasoning and a diff for each file. **Approve and deploy** commits the edits together with the issue's regression evals to the production branch, or as the next version of hosted source, deploys that commit to production, and links the deploy to the issue as its fix. It needs the same permissions as approving a proposal, and it offers the same choice of branch: on a GitHub or GitLab source, the fix can go to a separate branch for a pull request instead, without a deploy. With manual promotion, the deploy builds without serving traffic until it is made live. **Discard** drops the draft and returns to the first step.
3. **Confirm resolved.** While the fix deploy is built but not yet live, the issue shows that and offers **Make live**. Once it is live, check its runs, then **Mark resolved** closes the issue. If the fix deploy failed or is the wrong one, **Link another deploy** replaces it with a different production deploy.

A waiting fix is also listed under **Proposals**, where it can be approved or discarded the same way. If a file changed after the draft was written, approving fails and a new draft is needed. A fix for an issue that is resolved, dismissed, or merged cannot be approved until the issue is reopened.

A fix committed to a separate branch, or one whose deploy did not start, shows "not deployed", and the steps stay on **Review and approve**. Merge the branch into the production branch, deploy it, then use **Link fix deploy**. **Draft a new fix** starts over instead.

When the fix reached production another way, such as a merged pull request, use **Fixed another way?** under the steps: **Link fix deploy** links that production deploy as the fix, and **mark resolved** closes the issue directly.

Regression evals

Approving a fix turns up to five of the issue's runs into eval cases under `evals/`. Each case replays the user's request and is marked for review, because the recorded reply is the failure and must not become the expected answer. Add the assertions that describe the correct behavior before relying on the case. See Evals.

Approving preserves existing eval files and their assertions. Only missing cases are written.

PLATFORM

Proposals

A proposal is a source change waiting for review. It comes from one of three places:

- A deployed agent suggests a change to its own source after someone asks it in chat, such as "from now on, confirm before refunding". The agent cannot change itself.
- The builder chat in Studio suggests a change to any source file for the person building the agent.
- A fix drafted from an issue. It shows "Fix for issue" with a link to the issue.

Each proposal waits on the **Proposals** page in Studio until a person approves or discards it, and approving it commits the change and deploys it.

Turn it on

The `propose_source_change` tool is available to agents deployed on Amodal, in web chat and API chat. It is not registered under `amodal dev` or in channel sessions such as Slack.

Declare it beside `list_repo_files`, which the agent uses to see which files exist, and `read_repo_file`, which it uses to read a file before replacing it:

```
{ "tools": ["list_repo_files", "read_repo_file", "propose_source_change"] }
```

- **Default chat:** put the declaration in `agents/main/agent.json`. That file is read only when `agents/main/AGENT.md` exists and is not empty.
- **A named agent:** add both names to the `tools` list in `agents/<name>/agent.json`.
- **A new blank agent:** tick **Let the agent propose changes to itself** when creating it. Amodal writes both `agents/main` files.

Commit and deploy the change. The deploy must record its source version; otherwise the tool reports that the agent must be redeployed first.

The **Proposals** page is always in the Studio sidebar. Studio shows the builder chat only when the agent's active production deployment declares `propose_source_change` for its main agent or for one of its named agents, in `agent.json` or, for a named agent, in `agent.ts`, or when the agent already has a proposal from chat.

What an agent can change

A proposal replaces whole files. It can change or add:

- Markdown under `agents/`, `skills/`, and `knowledge/`, including the agent's prompt, `agents/<name>/AGENT.md` (`agents/main/AGENT.md` for default chat)
- `.ts`, `.mts`, `.js`, and `.mjs` files under `tools/<name>/`

The runtime reads `skills/`, `knowledge/`, and `tools/` from `amodal/<dir>/` when that directory holds files, and from the top level otherwise. A proposal must use the location this agent loads; one for the other location is

refused with the path to use instead. Every other path is refused, including `amodal.json`, `agent.json`, `agent.ts`, `tool.json`, `package.json`, connections, and environment files. An agent therefore cannot add a new tool, because a tool needs a `tool.json`, and cannot widen the connections or secrets it reaches.

Wherever `propose_source_change` is registered, the agent also gets `read_amodal_docs`, which reads these docs from docs.amodalai.com: the page index from `llms.txt`, or one page by its path, such as `/guide/connections`. The agent looks up the source format there instead of guessing it. Asked for something it cannot propose, such as a Slack connection, the agent tells the user how a person sets it up, from the docs, and does not propose a placeholder skill. It can still propose instructions that use the connection once it exists.

A skill added by a proposal reaches a named agent that lists its skills in `agent.json` only after someone adds it to that list.

Limit	Value
Files per proposal	10
Characters per file	200,000
Characters in the reason	2,000
Proposals waiting for review	10 per agent
Proposals listed on the page	50 per page, under Waiting and Decided

Amodal reads each file's current text itself, at the version the deployment runs, so the diff a reviewer sees is the platform's own. Files left unchanged are dropped, and a proposal that changes nothing is refused.

Builder chat

The builder chat is the chat button at the bottom right of every Studio page of an agent that shows it, as described in Turn it on. Use it to build an agent from Studio alone, starting from a blank agent created with **Let the agent propose changes to itself**: describe a connection, a policy, a hook, an eval, a store, a skill, or a custom React runtime app, and it proposes the files.

- **Who can use it:** anyone with permission to edit the agent's source. The button is hidden from everyone else, and from every agent that neither declares `propose_source_change` nor has a proposal from chat.
- **What it reads:** every source file at the current source version. Files that do not fit its budget of 160,000 characters are listed by path only. You can also paste or attach up to 5 images (about 2.5 MB total) per message when the configured model supports vision (`Anthropic`, `Google`, or `OpenAI`).
- **What it can propose:** creating, replacing, or deleting any source file except `.env` files, keys, and anything under `node_modules`, `dist`, `build`, `coverage`, `.git`, or `.next`. It cannot change a file too large to read, or delete a file it could not read. The diff of a deleted file shows every line removed, and its path is marked as deleted.
- **Secrets:** it never writes a secret value. A connection references a credential as `env:NAME`, and the builder tells you to add `NAME` on the **Secrets** page. A `spec.json` that holds a literal credential is refused.
- **Format:** it writes against the pages of these docs that describe the source layout, connections, hooks, evals, stores, knowledge, skills, tools, and custom runtime apps, and Amodal checks each connection, store, and

eval file and every JSON file, and that a new file sits where this agent loads its directory from, before storing the proposal. When a file does not pass, the builder gets one chance to fix it; otherwise it replies that nothing was proposed and names the problem.

- **Model:** the model set for issue triage and draft fixes on the Issues page, which is the agent's own model by default. Its calls are billed to the organization.

One chat keeps one waiting proposal. Each reply that writes files replaces it with a new proposal that holds every file so far, and the previous one moves to **Decided** as discarded. To take a file back out, ask the builder to drop it. A file that changes in the repository after the builder proposed it is left out of the next proposal, and the builder sees the repository's version of it. A reply that finishes after its waiting proposal was approved or discarded proposes nothing and says so; ask again to build on the current source. Resetting the chat starts a new proposal and leaves the previous one waiting. The conversation is not saved.

The builder chat runs on the hosted platform only, not under `amodal dev`. Its proposals are approved like an agent's, as described below.

Review a proposal

The agent replies with a link to **Deploy → Proposals** in Studio. Each waiting proposal shows the agent's reason, a diff for each file, the model, who asked for it, and **View chat**, which opens the conversation that asked for it. A proposal from the builder chat shows "From the builder chat" instead of **View chat**. The requester is the email of the user whose chat produced the proposal: the builder chat's user, or the owner of the agent chat. It shows the user id for someone outside the organization, and nothing when the chat ran on a token minted with an API key. A waiting proposal with a file that changed since it was proposed names that file and offers only **Discard**, because approving it would fail; ask for the change again. The page lists waiting proposals under **Waiting** and approved or discarded ones under **Decided**, with the count of each on its tab; an approved one shows its commit and deploy. The sidebar shows the number waiting next to **Proposals**.

- **Approve and deploy** needs permission to edit source and manage deploys: an organization admin or developer, or the agent's owner or editor.
- **Discard** needs permission to review feedback, which the same roles hold.

Approvals and discards appear in **Logs → Audit**.

What approving does

Approving asks for confirmation. On a GitHub or GitLab source, the dialog also offers a branch and a commit message:

- **Production branch**, the default, commits the change and deploys it, as described below.
- **New** creates a branch from the production branch and commits the change to it. **Existing** commits the change to another branch you pick. Neither deploys. The success message links to the page that opens a pull request on GitHub or a merge request on GitLab. The proposal moves to **Decided** with its commit marked "not deployed".

An Amodal-hosted source has no branches, so approving always commits the next source version and deploys it.

Approving onto the production branch:

1. Checks that every file still holds the text the agent read. If one changed, approving fails and the agent needs to propose the change again. Approving onto an existing branch checks the files on that branch instead.
2. Commits the change.
   - On a GitHub or GitLab source, the commit goes onto the production branch: the branch set in **Deploys**, or the repository's default branch when none is set.
   - On an Amodal-hosted source, the commit is the next source version.
3. Deploys that commit to production. When **Deploys** builds on push, the push and the approval share one build of the commit, whichever starts it first, and the proposal links to that build. It goes live when **Deploys** promotes automatically; otherwise make it live from its deploy page.

An issue fix also commits the issue's regression evals in the same commit, and links the deploy to the issue as its fix, which **Logs → Audit** records as a link fix entry on the issue.

If the commit succeeds but the deploy does not start, the proposal stays approved and the error names the commit. Deploy it from **Deploys**.

PLATFORM

Providers

Model credentials, allowed models, and model selections are platform settings. `amodal.json` has no model field.

Organization Providers

Organization > Model Providers lists the providers in the platform catalog, such as Anthropic, OpenAI, Google, DeepSeek, Groq, Mistral, and xAI. For each provider you can:

- Enable or disable it.
- Choose a credential source, such as your own key or Amodal-managed keys when the plan includes them.
- Add API keys or base URLs where the provider requires them.
- Select the models available to the organization.
- Verify the credentials.

A listed model is not proof of working upstream access. The provider can still reject it as unavailable or unauthorized, so verify credentials and run an eval through the deployed agent.

Agent Model

The agent's **Models** page sets the model, its effort level where supported, the thinking budget, the output-token limit, and the models bound to the `simple` and `advanced` tiers. Without a selection, the platform default is `google` / `gemini-3.5-flash`.

Hosted runtimes receive the selected model from the platform at deploy resolution. `amodal dev` accepts `--provider` and `--model` overrides.

Tool Schema Compatibility

OpenAI, Anthropic, and Google have separate SDK adapters. Other OpenAI-compatible providers use the OpenAI adapter with a configured base URL. Not every provider accepts every JSON Schema feature.

Native OpenAPI connections send each visible operation's schema as a tool definition. On the Google provider path, the model-facing copy simplifies closed-tuple schemas, because the Google SDK drops their positional item definitions. The runtime validates the actual arguments against the original schema before execution.

Compatibility depends on the provider route as well as the model. Use evals to check the route and schemas your agent uses, including write approval and denial.

PLATFORM

Authentication

A hosted agent has two access decisions: whether a visitor can open the app, and whether a request can use the runtime.

Setting	What it controls
`hosted_auth_mode: "user_auth"`	Team-only app access. Visitors sign in as members of the agent's organization. This is the default.
`hosted_auth_mode: "public"`	Anyone can open the app. Chat, sessions, and stores still require runtime authentication.
`end_user_auth`	End-user identity configuration. Hosted runtime support depends on the selected mode; see Identity settings.

Set these fields in **Settings > Auth** or with `PUT /api/agents/{agent_id}`, which requires an `admin` automation token. They belong to the platform agent record, not `amodal.json`. Redeploy after changing the runtime identity configuration.

Hosted browser login

For a team-only app served at the agent's own origin, including a custom React app:

1. Open the agent URL. The edge redirects an unauthenticated visitor to `/auth/login`.
2. Sign in through the hosted login flow. The edge checks organization membership and sets an HttpOnly `amodal_session` cookie.
3. Make same-origin runtime calls. The edge validates the cookie and adds the bearer header before forwarding the request.

Use `window.location.origin` as the SDK URL and omit `getToken` for this flow:

```
import { AmodalProvider, AmodalChat } from '@amodalai/react';
import '@amodalai/react/style.css';

<AmodalProvider runtimeUrl={window.location.origin}>
  <AmodalChat />
</AmodalProvider>;
```

The browser cannot read the HttpOnly cookie. `/auth/session` returns user identity, not a bearer token. Login requires a full-page navigation; an API fetch cannot complete the hosted login redirect.

API clients

CLI tools, CI jobs, and application servers send `Authorization: Bearer <credential>`:

Credential	Purpose	How to create it
Short-lived runtime token	Expiring, agent-bound access for runtime calls. Can include `scope_id` and `context` claims. The platform API does not accept it. Without `scope_id` or `role` claims, the runtime treats its holder as an operator.	Call `POST /api/agents/{agent_id}/tokens` from your server, for example with an operator/admin `pk_` token.
`ak_` agent API key	Persistent access for a service calling the runtime.	Create a key in the agent's API Keys settings. Keep it on your server.
`pk_` automation token	Platform operations such as configuring and deploying agents.	Create an organization automation token. Exchange it for a runtime token before calling chat.

The edge accepts agent-bound platform runtime tokens and `ak_` keys on team-only agents. You can run API smoke tests without opening the app to the public.

Keep persistent credentials out of browser code. A public app that offers chat needs a backend to authorize runtime access; setting `public` does not issue anonymous chat tokens.

Embedding in another product

For a multi-tenant product, proxy runtime calls through your backend and keep runtime credentials there. See Embedding & Multi-tenancy for the required checks.

When you use both `AmodalProvider` and `AmodalChat`, pass `getToken` to each. `AmodalChat` inherits only the provider's URL.

Identity settings

The Auth page stores one end-user identity configuration in `end_user_auth`:

Type	Fields	Where it applies
`none`	Optional synthesized user ID.	No end-user identity provider.
`jwks`	Application client ID, and optional JWKS URL, issuer, and audience.	Hosted chat runtime and Platform API. The platform supplies the JWKS URL and issuer when they are blank.
`jwt_secret`	Shared secret, `HS256` / `HS384` / `HS512`, optional issuer and audience.	Platform API only. The secret is used as saved; `env:` references are not resolved.
`oidc`	Issuer URL and client ID.	Platform API only. Keys come from `<issuer>/.well-known/jwks.json`, and the client ID is the audience.
`header`	Header name.	Platform API only. See below.
`custom`	Module path.	Stored only. The Platform API rejects it.

The OIDC form also shows client secret, callback URL, scopes, session secret, cookie, and TTL fields. They are stored but not used.

Hosted chat

The hosted chat runtime installs the `jwks` verifier when the client ID, issuer, JWKS URL, and audience are all known. The audience defaults to the client ID. The verifier checks issuer and audience, then checks agent membership. It replaces the platform JWT verifier for that runtime, so platform runtime tokens stop working there; `ak_` keys still work.

With any other type, hosted chat uses the platform JWT verifier. Saving `jwt_secret`, `oidc`, `header`, or `custom` does not change how hosted chat authenticates.

On a team-only origin, the edge still requires its hosted session, an `ak_` key, or an agent-bound platform runtime token. An external JWT does not bypass that gate.

Platform API

The Platform API accepts an end-user JWT only when the token has a `sub` claim and either an `agent_id` claim naming the agent or, for `oidc`, an `iss` matching the configured issuer. The `sub` becomes the end-user ID. An end-user JWT can reach only session and per-scope secret routes. Session routes return only sessions the user owns, and secret routes require the scope to equal its `sub`.

`header` does not authenticate a request. On a request already authenticated for the agent, the Platform API reads the end-user ID from the configured header, or `X-End-User-Id` when the name is blank. Use it only behind a proxy that strips client-supplied copies of that header.

PLATFORM

Embedding & Multi-tenancy

Use a scope to separate an agent's work by tenant, user, workspace, or record.

Value	Purpose
`scope_id`	Stable application identifier used to partition chat-session memory, stores, and credentials.
`session_id`	One conversation. Send it to continue that conversation.
`context`	Application facts for the turn. The React widget calls this prop `scopeContext`.

Several conversations can share one scope. Reusing a scope does not resume a conversation.

Authenticate the scope

A runtime token with a `scope_id` claim pins its caller to that scope: history, session reads, resumes, and session actions see only that scope's sessions, and a `session_id` from another scope starts a separate session. A token without a `scope_id` claim can filter history by any scope and resume any of the agent's sessions. A scoped token alone does not isolate tenants: the direct store routes ignore it, and the scope does not decide which of your users may act in it.

For a multi-tenant product, route runtime calls through your application backend:

1. Authenticate each client request with your application's login.
2. Authorize the requested scope and operation. For an existing session, check that the user can access that session in that scope. Apply these checks to chat, history, confirmations, and session actions.
3. With the default platform JWT verifier, mint a short-lived runtime token on your backend using `POST /api/agents/{agent_id}/tokens`. Include the authorized scope and application facts. `ttl` is in seconds, with a default of 3600 and a maximum of 86400. The response is `201` with `token` and `expires_at`. `context` values must be strings.

```
{
  "ttl": 300,
  "scope_id": "tenant:acme",
  "context": {
    "tenant_id": "acme",
    "plan": "premium"
  }
}
```

4. Keep the runtime token on your backend and use it to forward the authorized request. Record each created session's scope and ownership before allowing it to be resumed.

5. Expose only the methods and paths your application needs. Set history filters on the backend and check session ownership before reads, changes, deletions, and actions. Reject unsupported paths.

Returning a runtime token to the browser lets the caller bypass your proxy's checks. Tokens minted by an automation token do not establish a separate product-user identity. A product JWT signed with your own secret does not work on a hosted chat URL. See Authentication for credential types and identity settings.

Use stable identifiers such as `tenant:acme` or `case:merchant-123`, not display names. Renaming a customer should not change its data partition.

Claim and request precedence

Chat requests can also supply scope and context:

```
{
  "message": "Show open support tickets",
  "scope_id": "tenant:acme",
  "context": { "page": "support" }
}
```

A verified token's `scope_id` claim overrides the request body's value. Without that claim, the runtime accepts the body's scope, except that an end-user session token can name only the empty scope, its own user ID, or `<user ID>/<suffix>`, where the suffix is 1 to 128 characters from `A-Za-z0-9:_.-`. Use the last form to separate state per record, such as `user-123/taxpayer:TP-1001`. Any other scope fails with `403 SCOPE_FORBIDDEN`. Your backend must authorize and set the scope on every forwarded request.

Context merges per key: claim values win on conflict, and other body keys remain. Those body values are untrusted. Do not use the merged `scopeContext` alone to authorize an operation.

React integration

Point the widget at your application proxy. The proxy serves the runtime paths used by the widget and preserves streaming responses. It authenticates the application token returned by `getAppToken` and adds its own runtime credential to authorized upstream calls.

```
import { ChatWidget } from '@amodalai/react/widget';
import '@amodalai/react/widget/style.css';

function AgentPanel({ tenantId, getAppToken }: { tenantId: string; getAppToken: () => Promise<string> })
{
  return (
    <ChatWidget
      key={tenantId}
      serverUrl="https://app.example.com/api/agent"
      scopeId={`tenant:${tenantId}`}
      scopeContext={{ tenant_id: tenantId }}
      getToken={getAppToken}
    />
  );
}
```

The application token authenticates the browser to your backend. The `user` widget prop does not send an identity. A `key` tied to the scope remounts the widget when the tenant changes.

Continue a conversation

A chat request without `session_id` starts a session. To keep one conversation across navigation, save its ID and pass it back when the panel remounts:

```
import { useState } from 'react';
import { ChatWidget } from '@amodalai/react/widget';

function ScopedChat({
  userId,
  scopeId,
  getAppToken,
}: {
  userId: string;
  scopeId: string;
  getAppToken: () => Promise<string>;
}) {
  const storageKey = `amodal-session:${userId}:${scopeId}`;
  const [resumeSessionId] = useState(() => localStorage.getItem(storageKey));

  return (
    <ChatWidget
      serverUrl="https://app.example.com/api/agent"
      scopeId={scopeId}
      getToken={getAppToken}
      resumeSessionId={resumeSessionId ?? undefined}
      onStateChange={({ sessionId }) => {
        if (sessionId) localStorage.setItem(storageKey, sessionId);
      }}
    />
  );
}
```

Mount `ScopedChat` with a key containing the signed-in user and scope, because its state initializer runs only on mount. Clear saved session IDs on logout when a browser is shared. The backend must authorize the saved session before forwarding a resume or history request.

For a custom chat interface, read the session ID from the stream's `init` event. Send it as `session_id` to `streamChat`, or as `sessionId` in `RuntimeClient.chatStream` options. Load prior messages with `getSessionHistory`. Use the proxy URL and application authentication for these calls too.

Scoped resources

Resource	Chat-session behavior
Sessions	Labeled with a scope. A token with a `scope_id` claim lists and resumes only that scope's sessions; the backend authorizes access within a scope.
Memory	Stored and recalled within the session's scope.

Resource	Chat-session behavior
Stores	Agent store tools use the session's scope. Stores marked `shared` use the global partition and are read-only through scoped tools.
Credentials	`scope:KEY` resolves a secret for the current scope.

Without a scope, chat uses the empty-string partition, which does not separate tenants. The hosted runtime does not enforce `scope.requireScope`; see Scope.

The direct `/api/stores` routes and React store hooks use the scope that the token's `scope_id` claim pins, as agent store tools do. A token without that claim, and every store marked `shared`, uses the global partition. They do not inherit `ChatWidget.scopeId`, so a multi-tenant data UI needs tokens that pin each tenant's scope.

Scope context in connections and tools

A connection's `contextInjection` copies scope context values into outbound requests, such as a `tenant_id` into an `X-Tenant-Id` header. A custom tool reads the same values through `ctx.scopeContext`. Client-supplied keys are merged into this context, so use it for application facts such as the current page, and enforce tenant authorization in the receiving API.

Per-scope credentials

Reference a scope-specific secret in a connection:

```
{
  "auth": { "type": "bearer", "token": "scope:USER_API_TOKEN" }
}
```

The credential resolver looks up `USER_API_TOKEN` for the session's scope. In hosted deployments, manage scoped secrets through `PUT /api/agents/{agent_id}/scopes/{scope_id}/secrets`. Keep this operation on your backend.

`amodal dev` does not resolve `scope:` references, and a custom runtime host must supply its own credential resolver for them.

PLATFORM

Deployment API

The API reference lists every endpoint, the token kinds it accepts, and its request body.

Credentials

Deployment calls take a `pk_` automation token. An `ak_` agent API key cannot control deployments.

Token kind	Deployment use
`observer`	Read deployments, status, snapshots, build logs, and the active deployment.
`operator`	Build, activate, promote, roll back, start or stop runtimes, and change deploy settings.
`admin`	Setup that also creates agents, connects GitHub, or manages secrets.

Prefer an agent-bound `operator` token for routine releases. See Automation Tokens for creation rules.

The examples use these environment variables:

```
export AMODAL_API_BASE=https://api.amodalai.com
export AMODAL_AGENT_ID="YOUR_AGENT_ID"
# Set AMODAL_PLATFORM_TOKEN from your secret manager.
```

Build

A deployment builds the agent's selected source. For a connected GitHub or GitLab repository, push to the branch you want to deploy. For Amodal-hosted source, save a source version first.

Start a build with `POST /api/deployments/redeploy`:

```
curl --fail-with-body "$AMODAL_API_BASE/api/deployments/redeploy" \
  -H "Authorization: Bearer $AMODAL_PLATFORM_TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"agentId":"'"$AMODAL_AGENT_ID"'","branch":"main","environment":"preview"}'
```

`branch` defaults to the source's default branch. Send `commitSha` to build that commit of `branch` instead of its head. `environment` defaults to the environment a push of `branch` builds when `commitSha` is set, and to `production` otherwise. The response returns `deployId` and `buildId`. Poll `GET /api/deployments/{deployId}` for deployment status and `GET /api/builds/{buildId}/status` for build progress. Read build logs at `GET /api/builds/{buildId}/logs`.

Send `"reuseBuild": true` to share a build of the same commit: when the environment's newest build is that commit and has not failed, or a push is building it, the response returns that deployment with `reused: true` and starts no other build. Without `reuseBuild`, every call starts a build.

A completed build and an active deployment are separate states. Check runtime readiness and `isActive` before you continue. A build for any environment other than `production` activates when it completes. A `production` build activates only when automatic production promotion is on; otherwise it stays inactive until you activate it.

Deploy Settings

`GET` and `PUT /api/agents/{agent_id}/deploy-settings` read and change the settings that the Deploys page shows:

Field	Default	Behavior
`productionBranch`	Default branch	A push to this branch builds the `production` environment. A push to another branch builds `preview-<branch>`.
`autoDeploy`	`false`	When `false`, a push records source only and builds nothing.
`autoPromoteProduction`	`false`	When `true`, a completed `production` build activates automatically.

`ci.deploy` in `amodal.json` further limits which pushes build. See CI Rules.

A push builds nothing when the environment's newest build is already the pushed commit and has not failed, or when another push or a `reuseBuild` redeploy is building it.

Test And Activate

Test the deploy URL before changing production. Mint a short-lived runtime token with `POST /api/agents/{agent_id}/tokens` and use it as the bearer token for `/chat/stream`. A `pk_` token cannot authorize runtime chat. See Authentication.

`POST /api/deployments/{deployId}/activate` activates a deploy in its own environment and deactivates the previous one there. It does not copy a preview deploy into production.

Promote Between Environments

`POST /api/deployments/promote` copies the **active** deployment of one environment into a separate deployment in another environment and activates it:

```
curl --fail-with-body "$AMODAL_API_BASE/api/deployments/promote" \
  -H "Authorization: Bearer $AMODAL_PLATFORM_TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"agentId":"'"$AMODAL_AGENT_ID"'","fromEnvironment":"preview","toEnvironment":"production"}'
```

`toEnvironment` defaults to `production`. Confirm that the source environment's active deployment is the one you tested. If its runtime bundle is missing or cannot be copied, promotion fails without changing the destination. Promotion activates the destination regardless of `autoPromoteProduction`.

For GitHub source, a pending or failing `amodal/evals` commit status blocks promotion with `422 EVAL_GATE_BLOCKED`. A missing status does not block it. Pass `"force": true` to override the check. See Evals.

Roll Back

Pass the deployment ID to activate in the target environment:

```
curl --fail-with-body "$AMODAL_API_BASE/api/deployments/rollback" \
  -H "Authorization: Bearer $AMODAL_PLATFORM_TOKEN" \
  -H "Content-Type: application/json" \
  -d
'{"agentId":"'"$AMODAL_AGENT_ID"'","deployId":"'"$KNOWN_GOOD_DEPLOY_ID"'","environment":"production"}'
```

Without `deployId`, rollback selects the most recently activated inactive deployment that completed and was live before. If none qualifies, rollback fails without changing the active deployment. Use an explicit ID for a verified release.

Runtime URLs

URL	Behavior
`https://agent-slug.amodalapp.com`	Persistent agent URL for the active production deployment.
`https://deploy-id--agent-slug.amodalapp.com`	URL pinned to one deployment.

Activation, promotion, and rollback in production change where the persistent URL points. A deploy URL stays pinned to its deployment while that deployment and its runtime are retained.

Platform API

Use the Platform API to create, deploy, and operate agents from scripts and external tools. The base URL is `https://api.amodalai.com`.

Authenticate with a platform automation token:

```
Authorization: Bearer pk_...
```

Each operation lists the token kinds it accepts: `observer`, `operator`, or `admin`. A platform automation token does not authorize runtime chat; mint a runtime token with `POST /api/agents/{agent_id}/tokens`.

Download The Reference

The docs build generates the reference from the Platform API's endpoint metadata:

```
curl -fsSL https://docs.amodalai.com/openapi.json -o amodal-openapi.json
```

The OpenAPI document covers supported automation endpoints, accepted token kinds, and documented parameters and request bodies. Its success responses use generic object schemas. Check the actual HTTP status and returned JSON when handling a response.

See Build With A Coding Agent for setup and Deployment API for a release workflow.

Interactive OpenAPI reference available at https://docs.amodalai.com/api

SDK Overview

`@amodalai/react` connects a web application to a deployed Amodal runtime over HTTP and server-sent events.

Need	Use
Add chat to a React app	`ChatWidget`, available from `@amodalai/react/widget`.
Build a UI with agent actions, sessions, and events	`AmodalProvider` and hooks.
Render inline chat inside a provider	`AmodalChat`.
Call runtime APIs from imperative code	`RuntimeClient`, `SdkClient`, or `ChatClient`.

Install

```
npm install @amodalai/react
```

The package requires React and React DOM 18 or 19. Import its stylesheet when using the chat components.

Hosted React app

For an app served on the agent's origin with the default team login:

```
import { AmodalProvider, AmodalChat } from '@amodalai/react';
import '@amodalai/react/style.css';

export function App() {
  return (
    <AmodalProvider runtimeUrl={window.location.origin}>
      <AmodalChat />
    </AmodalProvider>
  );
}
```

The hosted cookie authenticates same-origin calls. Use direct runtime access when users are trusted to access the agent's sessions across scopes.

For a multi-tenant product, point the SDK at an application proxy that retains runtime credentials and authorizes each scope, session, and operation. Return your application token through `getToken`, passing the callback separately to the provider and chat component. See Authentication and Embedding.

REACT SDK

@amodalai/react

React bindings for a deployed Amodal runtime:

1. **Provider + hooks**: `AmodalProvider` gives every hook a configured client; hooks read stores, invoke tools and skills, follow sessions, and subscribe to live events.
2. **Chat UI**: `ChatWidget` (and its preset `AmodalChat`) renders a full streaming chat with tool calls, confirmations, and widgets. Detailed on the Chat page.
3. **Headless clients**: `SdkClient`, `RuntimeClient`, and the SSE/chat-api utilities, for when a hook doesn't fit (imperative code, non-React state layers).

Installation

```
npm install @amodalai/react
```

AmodalProvider

Wrap your app to configure the clients used by provider-based hooks:

```
import { AmodalProvider } from '@amodalai/react';

<AmodalProvider runtimeUrl={window.location.origin}>{children}</AmodalProvider>;
```

Prop	Required	Meaning
`runtimeUrl`	yes	Base URL of the agent's runtime server.
`getToken`	no	Returns a bearer token for authenticated requests (sync or async).
`metadata`	no	Observability-only values sent with chat requests from provider-based hooks and `AmodalAction`; a map, or a function read on each send. See Request metadata.

For an app served by Amodal at the agent's own origin, use a same-origin `runtimeUrl` such as `window.location.origin` and **omit** `getToken`: hosted login is cookie-backed and the bearer is injected at the edge. For direct bearer authentication by callers trusted across the agent's scopes, return a runtime-compatible token from `getToken`. Multi-tenant apps use an application proxy as `runtimeUrl` and return an application token instead. `AmodalChat` only inherits the provider's URL: pass its `getToken` separately. See Authentication for supported token types.

`useAmodalContext()` returns the provider's internals (`client: RuntimeClient`, `sdk: SdkClient`, `runtimeUrl`, `getToken`) for anything the hooks don't cover; it throws outside a provider.

Request metadata

Every chat request from the SDK carries `metadata` that tells Amodal where the message came from. It is recorded on the run's trace and the session and is never sent to the model. Amodal shows it on the trace, session, and feedback pages, and filters runs, sessions, and feedback by `userId` and `screen`.

The SDK collects these keys from the browser:

Key	Value
`screen`	Page path, without the query string or fragment.
`pageTitle`	`document.title`, when set.
`referrer`	Referrer origin and path, without query or fragment.
`userAgent`	`navigator.userAgent`.
`locale`	`navigator.language`.
`timezone`	The browser's IANA time zone.
`viewport`	Window size as `<width>x<height>`.

Add what only your app knows through the `metadata` prop of `AmodalProvider` or `ChatWidget`. `ChatWidget` and `AmodalChat` send only their own `metadata` prop; the provider's value applies to provider-based hooks and `AmodalAction`. Your values replace the collected ones with the same key, so a readable `screen` name can replace the path. Pass a function to read current values on each message:

```
<AmodalProvider
  runtimeUrl={window.location.origin}
  metadata={() => ({ userId: currentUser.id, screen: currentScreenName(), plan: account.plan })}
>
  {children}
</AmodalProvider>
```

Values must be strings. A request accepts at most 50 keys, keys up to 64 characters, and values up to 1,024 characters. The runtime rejects a request over these limits, so the SDK drops any of your entries that breaks them, logs a console warning naming each one, and past 50 keys keeps yours over the collected ones. Any caller can send any value, so Amodal labels metadata unverified and shows the authenticated user beside it when the request carries one. Do not put secrets or sensitive personal data in metadata. The runtime reference describes the recorded attributes.

Reading and writing stores

These hooks call the runtime's direct store API. With a token whose `scope_id` claim pins a scope, they read and write that scope's rows, the same rows agent store tools use in that scope's sessions. A token without that claim, and every store declared `shared`, uses the global (empty-string) partition. The hooks do not inherit `ChatWidget.scopeId`. See Embedding.

useStoreQuery

```
const { data, error, isLoading, refetch } = useStoreQuery<Submission>('submissions', {
  where: { status: 'received' },
  orderBy: 'updated_at',
  order: 'desc',
  limit: 50,
});
// data: Array<{ key: string; value: Submission; updatedAt: string }>
```

`where` is an equality filter on payload fields; `orderBy` / `order` sort server-side. Without `limit`, the runtime returns 20 rows; set `limit` to read more. Limits above 500 are fetched in 500-row pages. Call `refetch()` after an action to pull in new rows.

useStoreEntry

Read one document by key and update its payload:

```
const { data, isLoading, mutate, refetch } = useStoreEntry<Submission>('submissions',
'sub_bistro_ember');
```

`mutate(next)` POSTs the payload and refetches. The runtime derives the document key from the payload's key template, so `next` must include its key field(s).

useStoreActions

Use `get` and `set` in event handlers:

```
const submissions = useStoreActions<Submission>('submissions');
await submissions.set('sub_x', row);
```

`set(key, value)` accepts the key for compatibility; the runtime derives the stored key from `value`. The exported `remove(key)` calls a DELETE endpoint that the runtime store router does not implement.

Writes through `mutate` or `set` to a store declared `shared` return `403` unless the caller has an operator credential.

Running work

useToolRun

Invoke a **durable tool** (`"execution": "durable"` with an `invoke` trigger) from a button or form. `run` resolves with `{ sessionId, outcome }`. Read the run status through `useSession`; tools can also write results to stores.

```
const analyze = useToolRun('analyze-submission');

await analyze.run({ submission_id });
// analyze.status: 'idle' | 'running' | 'succeeded' | 'failed'
// analyze.output.outcome, analyze.error, analyze.reset()
```

`run` resolves when the session reaches a terminal or operator-facing state: if the runtime answers synchronously with an outcome it is used directly; a queued start is watched (polled) to its outcome. Pass `scopeId` and `scopeContext` in the second argument to `useToolRun` when the tool operates on a scoped resource.

`useToolTrigger` enqueues through `POST /api/enqueue`, then watches the session. It has the same launcher interface as `useToolRun`. See Tools.

useAutomation

`useAutomation()` schedules a durable tool, subagent, or prompt. `schedule(target, { schedule, input?, label?, enabled? })` takes a tool name or `{ kind, ref }` and a schedule of `{ cron, timezone? }`, `{ every }` (`5m`, `15m`, `1h`, `1d`, or `1w`), or `{ at }`. It also returns `list`, `enable`, `disable`, and `remove`. `useAutomations()` lists the caller's automations; its `intent` filter is ignored.

useSkillAction

Send a chat prompt that asks the agent to use a declared skill, then collect its text response. The model chooses how to follow the prompt. Use a durable tool when the action requires a defined execution path.

```
const namer = useSkillAction('subject-line-writer');
namer.execute({ topic: 'renewal reminder', count: 3 }); // params are serialized into the generated prompt
// namer.result (string | null), namer.loading, namer.error
```

Worked example

This example displays global store rows and refetches them after a durable tool runs:

```
function SubmissionsScreen() {
  const submissions = useStoreQuery<Submission>('submissions', { limit: 200 });
  const findings = useStoreQuery<RiskFinding>('risk_findings', { limit: 200 });
  const analyze = useToolRun('analyze-submission');

  async function onAnalyze(id: string) {
    await analyze.run({ submission_id: id });
    await Promise.all([submissions.refetch(), findings.refetch()]);
  }

  if (!submissions.data?.length) return <EmptyState hint="send `seed` in chat first" />;
  return <SubmissionsTable rows={submissions.data} findings={findings.data} onAnalyze={onAnalyze} />;
}
```

Sessions

Use session hooks to display durable tool runs, scheduled work, and pending user input. The runtime history API hides sessions owned by another verified user and, when the token has a `scope_id` claim, sessions in other scopes. See Sessions. For a multi-tenant UI, point the provider at an application proxy that authorizes session access and retains runtime credentials.

Hook	What it gives you
`useSessions(query?)`	List the agent's sessions. The runtime ignores `query`.
`useSession(id)`	Full detail for one session: status, metadata (surface, current step, pending input request, progress), messages. Poll via `refetch`, or pair with `useAgentEvents` for live updates.
`useSessionActions(id)`	Operator actions: `answer(value, edits?)` resolves a pending `requestInput` (the runtime enforces eligible answerers), `cancel()`, `restart()`.
`useScheduledTools()`	Sessions fired by schedules (cron), for a "scheduled runs" list.
`useToolCallLog({ sessionId })`	Per-session tool-call audit log (options also accept `since` and `limit`).

Live events

`useAgentEvents` subscribes to the runtime's event bus over SSE (`/api/events`):

```
const { events, latest, isConnected } = useAgentEvents({
  include: ["tool_calls"],   // server-side filter: categories or event-type names
  sessionId,                 // optional client-side narrowing to one session
  onEvent: (e) => { ... },
});
```

Type predicates (`isToolCallStarted` , `isToolCallCompleted` , `isToolCallFailed`) narrow the discriminated `AgentEvent` union. Reconnects resume from the last delivered event id.

The hook opens a plain `EventSource` and does not send `getToken` . On a hosted runtime it therefore relies on the same-origin session, receives events without content fields such as `argsSummary` and `error` , and receives every event type regardless of `include` . See Event Bus.

Chat components

AmodalChat and ChatWidget

`ChatWidget` supports positions, history, theming, callbacks, and custom transports. `AmodalChat` wraps it with `position="inline"` and defaults `serverUrl` from the nearest `AmodalProvider` .

```
import { AmodalChat } from '@amodalai/react';

<AmodalChat />;
```

Props, positions, theming, and the widget's building blocks are documented on the Chat page.

AmodalAction

Fires a chat prompt from a button or other UI element:

```
import { AmodalAction } from '@amodalai/react';

<AmodalAction prompt="Summarize today's alerts" label="Get Summary" />;
```

`AmodalAction` asks the model to follow a prompt. `useToolRun` invokes a named durable handler directly. The handler may itself call a model; determinism depends on its implementation.

Confirmation and review cards

Chat renders approval controls for `confirmation_required` events. These components are also exported for standalone approval UIs:

- `ConfirmCard` : endpoint, method, reason, approve/deny.
- `ReviewCard` : the structured variant; full parameters, escalation badge, connection name.

Both take `{ confirmation, onApprove, onDeny }` .

`FormattedMarkdown` , a prose-styled Markdown renderer, is exported for rendering agent output outside the chat surfaces.

Chat hooks

For a custom chat surface without `ChatWidget` :

Hook	Description
`useAmodalChat`	Provider-based chat session: send messages, streamed state, confirmation callbacks.
`useChat`	The widget's chat state machine over `/chat/stream`, `serverUrl`-based.
`useChatStream`	The transport-agnostic reducer underneath `useChat`; you supply `streamFn`.
`useSessionHistory`	List, retag, rename, and delete previous chat sessions.

`useAmodalChat` and `useChat` prepare a session through `POST /sessions` when they mount and send its ID with the first message, so the first reply skips the session setup. A hook that resumes a session prepares nothing, and `reset` prepares the next session. If the runtime cannot prepare one, the first message starts its own session. A prepared session that is still unused is rebuilt in the background when the page regains focus after most of the runtime's idle timeout has passed, before the runtime evicts it. `useChat` and `ChatWidget` also rebuild it after the token returned by `getToken` changes. An application proxy can therefore see `POST /sessions` requests beyond the one at mount. Pass `prepareSession: false` to `useAmodalChat` where a mount rarely leads to a message; `AmodalAction` does.

When `keepAliveOnUnmount` is enabled (the default in `useChat` and `ChatWidget`), chat requests set `continue_on_disconnect: true` with a client `turn_id` so an active turn keeps running on the runtime if the view unmounts or disconnects, and a remounted or separate browser view resuming that session discovers and observes the active turn through `GET /chat/sessions/:id/turns/:turnId`.

`ArtifactHost` and `useArtifactHost()` share published files and registered artifacts between `ChatWidget` and surrounding application panels.

`useAmodalBrief`, `useAmodalInsight`, and `useAmodalQuery` each wrap one chat exchange through `RuntimeClient.chatStream`.

`RuntimeClient.startTask` and `useAmodalTask` call `/task` endpoints that neither the hosted runtime nor `amodal dev` serves. Use `useToolTrigger` with a durable tool for background work.

Headless clients

Everything the hooks do is available imperatively:

- `SdkClient` (via `useAmodalContext().sdk` or constructed directly): `runTool`, `triggerTool`, `watchSession`, session actions, store access via `.runtime`, files (`readFile`, `listDir`; hosted runtime only), automations.
- `RuntimeClient`: `chatStream`, store document reads.
- `ChatClient` / `ChatStream`: headless chat with typed events, no React.
- `streamChat`, `createSession`, `listSessions`, `getSessionHistory` (chat API), `streamSSE` / `parseSSELine` (SSE utilities): the lowest-level pieces, used by the clients above.

`@amodalai/react/client` exports `RuntimeClient`, `ChatClient`, `ChatStream`, and the chat API and SSE utilities without the components and hooks. `SdkClient` is exported from the package root only.

Unsupported endpoints

These call endpoints the runtime does not serve:

- `useAuth`, `useTeamMembers`, and `useMemory`. On a hosted runtime they return a `NOT_IMPLEMENTED` `SdkError` through their `error` field. Under `amodal dev` with the bundled chat UI, the error is `UNKNOWN`.
- `useConnections`, which requests a connection list endpoint that does not exist.
- `useStoreActions().remove`, described above.
- `SdkClient.writeFile`, which calls `PUT /api/files/*`. Neither the hosted runtime nor `amodal dev` serves that route.

Chat

`ChatWidget` renders streaming chat, tool calls, approvals, questions, and session history. `AmodalChat` wraps it with `position="inline"` and a runtime URL from the nearest `AmodalProvider`.

Import `ChatWidget` from `@amodalai/react/widget` for a chat-only integration, or from the package root when using provider hooks. The widget subpath also exports components and hooks for a custom chat interface.

Installation

```
npm install @amodalai/react
```

Quick start

Inside a same-origin `AmodalProvider` using the default hosted team login:

```
import { AmodalChat } from '@amodalai/react';
import '@amodalai/react/style.css';

<AmodalChat />;
```

For a multi-tenant app, use an application proxy that authenticates each request and authorizes scope and session access. The callback supplies your application token; the proxy retains runtime credentials:

```
import { ChatWidget } from '@amodalai/react/widget';
import '@amodalai/react/widget/style.css';

<ChatWidget serverUrl="https://app.example.com/api/agent" getToken={getAppToken} position="floating"
historyEnabled />;
```

Keep runtime tokens and persistent API keys on the backend for multi-tenant use. Direct runtime access is suitable only for callers trusted to access the agent's sessions across scopes. `AmodalChat` inherits the provider's URL only; pass `getToken` to it separately for bearer authentication. See Embedding for proxy requirements.

Props

`AmodalChat` accepts every `ChatWidget` prop except `position` (pinned to `inline`); its `serverUrl` is optional and defaults to the provider's `runtimeUrl`.

Connection and identity

Prop	Default	Meaning
`serverUrl`	required	Base URL of your application proxy, or the runtime for trusted direct access.
`user`	none	`{ id: string }`; accepted but neither sent to the runtime nor displayed. It does not authenticate a user.
`getToken`	none	Returns a bearer token for `serverUrl` (sync or async): an application token for your proxy, or a runtime token for trusted direct access.
`scopeId`	none	Scope requested for chat sessions, memory, and agent store tools. A verified token scope takes precedence. A hosted browser session without a scope claim may request only its own user ID; another value returns `403`.
`scopeContext`	none	Key-value pairs injected into connection API calls via `contextInjection`.
`metadata`	none	Observability-only values such as `userId` and `screen`; a map, or a function read on each send. See Request metadata.

`scopeId` identifies the tenant, workspace, or record; `sessionId` identifies one conversation. Reusing a scope does not resume a chat. Use `resumeSessionId` to continue a saved conversation. Your backend must authorize the scope and session on every request; see Embedding.

Behavior

Prop	Default	Meaning
`position`	`floating`	See Positions.
`defaultOpen`	`false`	Start open. Required for `right` and `bottom`, which have no launcher.
`historyEnabled`	`false`	Session history drawer.
`showHeader` / `showInput` / `showFeedback`	`true` / `true` / `false`	Header bar; input bar; thumbs up/down on assistant messages.
`autoFocus`	`false`	Focus the input when it mounts.
`agent`	none	Root-agent surface name from the bundle's agents; that agent's declared resources scope what the session loads.
`deployId`	active deploy	Pin a specific deployment.
`initialMessage`	none	Auto-sent once on mount.
`initialDraft`	none	Placed in the input on mount for the user to review and send.
`resumeSessionId`	none	Load an existing session on mount and continue sending on that session; takes precedence over `initialMessage`.
`persistKey`	none	`localStorage` key used to persist and restore `sessionId`.

Prop	Default	Meaning
`attachments`	`false`	Show the file/image attachment button in the input bar.
`showReasoning`	`true`	Display `thinking_delta` reasoning blocks.
`generateFollowUps`	none	`(messages) => string[] | Promise<string[]>` for suggested follow-up chips after a reply.
`keepAliveOnUnmount`	`true`	Keep an in-flight turn running on the runtime when the widget unmounts or disconnects, and reattach on resume.

Callbacks

Prop	Meaning
`onToolCall(call)`	A tool call completed; receives the full `ToolCallInfo`.
`onKBProposal(proposal)`	The agent proposed a knowledge-base update.
`onEvent(event)`	Every widget event (agent-driven and interaction).
`onSessionCreated(sessionId)`	First stream init returned a session id.
`onStreamEnd()`	The SSE stream ended.
`onStateChange({ sessionId, messages })`	Session state changed (for external persistence).

Extension points

Prop	Meaning
`theme`	See Theming.
`widgets`	`WidgetRegistry` of custom renderers for rich inline widgets.
`inlineBlockRenderers`	Renderers for block types the widget doesn't render natively; native types (text, widget, reasoning, tool_calls, confirmation, ask_user, ask_choice, durable_approval, show_preview) cannot be overridden.
`streamFn`	Custom transport (`(text, signal, images?, attachments?) => AsyncIterable<SSEEvent>`) for chat messages. The widget still calls `serverUrl` to prepare a session on mount and to load history. To refuse a message, throw `MessageNotSentError` from `@amodalai/react` before yielding the first event. The widget puts the text, images, and attachments back in the message box and shows the error's message to the user.
`onDurableApproval(sessionId, value, followUpMessage?)`	Required with a custom `streamFn` when handling durable approval cards. Submit the answer to the runtime and resume or follow the run. The standard transport does this itself.

Prop	Meaning
`askDisplay`	`inline` by default. Use `docked` when the host displays questions through `AskChoiceBanner` and durable approvals through `DurableApprovalBanner`, both from `@amodalai/react`.
`entityExtractors`	Replaces the default entity extractor for entity events.
`onOpenFile(file)`	Open a published file in the host's own layout. Without it, the widget shows the file in place of the transcript. Render the file with `FileViewer`.
`artifactProvider` / `artifactProviders`	Discovers and previews files stored by the embedding app (`ArtifactProvider` or a map keyed by `providerId`), including links and structured registrations in restored conversations. Pass `artifactAuthToken` when the provider needs a bearer token.
`onArtifactsChange(files)`	Receives the runtime's and the app's artifacts together.
`autoOpenArtifacts`	Defaults to `true`. Opens newly produced artifacts when a live turn finishes; restored history stays closed.

Imperative handle

The `ref` exposes `ChatWidgetHandle`:

```
const chat = useRef<ChatWidgetHandle>(null);
chat.current?.sendMessage("Summarize today's alerts"); // optional second argument: { attachments }
chat.current?.getSessionId();
```

Other handle methods: `stop()`, `isStreaming()`, `toggleHistory()`, `toggleArtifacts()`, `getArtifacts()`, `openFile(file)`, `closeFile()`, `submitAskChoice(askId, values, message)`, and `respondDurableApproval(sessionId, value, followUpMessage?)`.

Keeping A Scoped Chat Alive

To retain one conversation per user and scope, store the session ID and restore it when the panel remounts. Use your authorized proxy URL as `serverUrl`:

```
const storageKey = `amodal-session:${userId}:${scopeId}`;
const [resumeSessionId] = useState(() => localStorage.getItem(storageKey));

<ChatWidget
  serverUrl={serverUrl}
  getToken={getAppToken}
  scopeId={scopeId}
  scopeContext={{ tenant_id: tenantId, page: 'case-review' }}
  resumeSessionId={resumeSessionId ?? undefined}
  onStateChange={({ sessionId }) => {
    if (sessionId) localStorage.setItem(storageKey, sessionId);
  }}
/>;
```

The containing component must remount when the user or scope changes because `useState` reads storage only on mount. Give it a React `key` containing both values. Clear saved IDs on logout when the browser is shared. The backend must check access to the saved session; a storage key does not enforce it. See the complete embedding example.

Positions

Position	Behavior
`inline`	Renders in-place within your layout
`floating`	Floating button that expands into a chat panel
`right`	Fixed panel on the right side; needs `defaultOpen`
`bottom`	Fixed panel at the bottom; needs `defaultOpen`

Theming

The `theme` prop covers the common cases:

```
<ChatWidget
  serverUrl={serverUrl}
  getToken={getAppToken}
  theme={{
    mode: 'auto', // 'light' | 'dark' | 'auto' (follows prefers-color-scheme)
    primaryColor: '#6e56cf',
    borderRadius: '12px',
    headerText: 'Ask the agent',
    placeholder: 'Type a message…',
    verboseTools: true, // full tool-call params, results, timing
  }}
/>
```

Other `ChatTheme` fields: `backgroundColor`, `fontFamily`, `fontSize`, `userBubbleColor`, `agentBubbleColor`, `toolCallColor`, `emptyStateText`.

Which key drives which surface

Each `theme` key sets one CSS custom property that drives specific surfaces. `primaryColor` also colors the floating launcher through an inline style, so overriding `--pcw-primary` in CSS does not recolor the launcher.

`theme` key	CSS variable	Surfaces it controls
`primaryColor`	`--pcw-primary`	Send button, focus rings, accent text/links; launcher inline
`backgroundColor`	`--pcw-bg`	Panel and header background
`userBubbleColor`	`--pcw-user-bubble`	User message bubbles
`agentBubbleColor`	`--pcw-agent-bubble`	Agent bubbles and inline cards
`toolCallColor`	`--pcw-tool-call-bg`	Tool-call card background
`borderRadius`	`--pcw-radius`	Panel, bubbles, buttons
`fontFamily` / `fontSize`	`--pcw-font` / `--pcw-font-size`	All widget text
`headerText`	`--pcw-header-text` †	Header title label
`placeholder`	`--pcw-placeholder` †	Input placeholder
`emptyStateText`	`--pcw-empty-state-text` †	Empty-conversation message
`mode`	`data-theme` / `prefers-color-scheme`	Light vs. dark palette (see below)

† These variables carry a text string, not a style value, and the text is rendered by the component rather than read back from the variable. Set the prop to change the copy; overriding the CSS variable alone has no effect.

Surfaces with no matching `theme` key (header/body text, muted text, borders, panel tint, shadow) are CSS-only: `--pcw-text`, `--pcw-text-muted`, `--pcw-border`, `--pcw-panel-tint`, `--pcw-user-text`, `--pcw-shadow`. Override those directly.

For anything the prop doesn't cover, override the CSS custom properties (no Tailwind dependency):

```
.pcw-widget {
  --pcw-primary: #6e56cf;
  --pcw-bg: #ffffff;
  --pcw-text: #1a1a1a;
  --pcw-border: #e5e5e5;
  --pcw-radius: 12px;
}
```

Saved Embed settings

The agent's **Embed** page saves configuration for its preview and generates a React snippet. A mounted `ChatWidget` reads its props; it does not fetch that saved configuration.

After changing saved settings, copy the generated values into your application and release that change. Editing props in your application directly also works.

Building blocks

For a custom chat surface, the `/widget` subpath exports the pieces `ChatWidget` is made of:

- **Components**: `MessageList`, `InputBar`, `SessionHistory`, `StreamingIndicator`, `ToolCallCard`, `AskUserCard`, `KBProposalCard`, `SkillPill`, `TagEditor`, `FormattedText`, and `WidgetRenderer` (with its `WidgetRegistry` type).
- **Hooks**: `useChat` (the widget's chat state machine), `useWidgetEvents`, `useSessionHistory`.
- **Chat API**: `listSessions`, `getSessionHistory`, `createSession`.
- **Theme utilities**: `defaultTheme`, `applyTheme`, `mergeTheme`.

The provider-based hooks and headless clients live in the package root; see @amodalai/react.

REFERENCE

Runtime Server

The runtime executes agents and exposes chat, session, and tool APIs. Hosted deployments supply authentication and durable storage through the platform. The CLI runs a local server with in-memory storage.

Use the Platform API to manage agents, source, and deployments. Use the endpoints below to interact with a running agent.

Endpoints

Chat

Method	Path	Response
`POST`	`/chat` or `/chat/stream`	SSE stream of Amodal events
`POST`	`/chat/sync`	Complete response after execution finishes
`POST`	`/chat/ai-stream`	AI SDK streaming protocol

The ordinary chat SSE stream is a POST response. Use `fetch` or the SDK; browser `EventSource` supports GET only.

A chat request whose `session_id` already has a turn running returns `409 {error: 'turn_in_progress'}`. To send a message while a turn runs, use `POST /chat/sessions/:id/messages`.

Runtime auth from curl

Hosted chat requires a runtime bearer token or the hosted browser session flow. Public access to an app's static files does not grant access to chat.

With `AMODAL_API_BASE`, `AMODAL_AGENT_ID`, `AMODAL_PLATFORM_TOKEN`, and `AMODAL_AGENT_URL` set, mint a short-lived runtime token:

```
AMODAL_AGENT_TOKEN="$(
  curl -fsS "$AMODAL_API_BASE/api/agents/$AMODAL_AGENT_ID/tokens" \
    -H "Authorization: Bearer $AMODAL_PLATFORM_TOKEN" \
    -H "Content-Type: application/json" \
    -d '{"ttl":600}' \
    | node -e 'const fs = require("fs"); const body = JSON.parse(fs.readFileSync(0, "utf8"));
process.stdout.write(body.token)'
)"

curl -fN "$AMODAL_AGENT_URL/chat/stream" \
  -H "Authorization: Bearer $AMODAL_AGENT_TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"message":"Say hello."}'
```

The platform token needs permission to mint runtime tokens for that agent. Keep it on your backend. See Authentication for persistent agent API keys and browser authentication.

Chat request body

Field	Type	Description
`message`	string, required	User text. May be empty when `images` or `attachments` are supplied.
`session_id`	string	Continue this conversation; omit to start one.
`scope_id`	string	Tenant, user, or object scope. Scope requirements depend on the host; see below.
`context`	`Record<string, string>`	App facts for prompt templates and connection context injection. These are not trusted authorization claims.
`metadata`	`Record<string, string>`	Observability only: recorded on the turn's trace and the session, never sent to the model. See below.
`agent`	string	Named agent under `agents/`. Omit to use `default` when defined, otherwise the unscoped default chat.
`skill`	string	Select a skill-scoped session.
`deploy_id`	string	Deployment hint for a host that resolves bundles dynamically. It does not switch the deployment behind a fixed agent URL.
`model`	`{provider, model, effort?}`	Model selection for the session, subject to the host's provider access.
`max_session_tokens`	positive integer	Cumulative session token budget. Checked between transitions, so usage can exceed it during a model call.
`images`	array	Up to 5 image attachments.

Field	Type	Description
`attachments`	array	Files as `{mimeType, data, filename?, delivery?}`. `data` is base64 without a data-URI prefix.
`turn_id`	string	Client-supplied identifier for a resumable turn (`[A-Za-z0-9_-]{1,128}`).
`continue_on_disconnect`	boolean	Keep the turn running on the runtime if the SSE connection closes so a reconnecting client can observe or cancel it.

A scope can contain several sessions. Sending the same `scope_id` without `session_id` starts another conversation. Your application backend must authorize the scope and any requested session before forwarding the call; see Embedding.

Hosted chat does not enforce `scope.requireScope` from `amodal.json`. For hosted multi-tenant use, proxy calls through your backend, authorize each scope and session, and retain runtime credentials there.

A `scope_id` claim on the caller's token overrides the request's `scope_id`. The runtime then limits that caller's chat resume, session history, and durable session actions to the claimed scope. The direct store API is not partitioned by scope. A hosted browser session without a `scope_id` claim may request only the agent-level scope or its own user ID; any other `scope_id` returns `403`. Its automations under `/api/automations` use its own user ID in place of the agent-level scope, which holds the operator's automations. An automation runs with the authority of the user who saved it: an end user's runs get the end-user tool set and restrictions. An end user's automation is disabled at its next run once that user can no longer use the agent.

Attachments support PDF, plain text, Markdown, CSV, PNG, JPEG, GIF, WebP, Office Open XML, OpenDocument, and RTF. Office, OpenDocument, and RTF files are converted to text at ingestion. Each PNG, JPEG, GIF, or WebP image embedded in an Office or OpenDocument file follows its text as a separate image attachment named `<filename>/<image>`, up to 20 images and 15 MB across all attachments in the request, each image at most 3.75 MiB. The converted text lists the attached images and any embedded media left out, with the reason.

`metadata` describes where the request came from, such as `{"userId": "u_42", "screen": "/billing"}`. It accepts at most 50 keys, each up to 64 characters, with string values up to 1,024 characters; a request over these limits is rejected with `400`. The runtime records each key on the turn's root span as `amodal.meta.<key>` and keeps the latest value of each key on the session. The model, tools, prompt templates, and guardrails never see it. Any caller can set any value, so Amodal labels it unverified. The authenticated user, organization, and sign-in method are recorded separately as `amodal.user.id`, `amodal.user.org_id`, and `amodal.auth.method`, and `metadata` cannot override them. Amodal shows the email of an authenticated user who is an active member with read access to the agent's organization or an active collaborator on the agent. Amodal filters runs, sessions, and feedback by the `userId` and `screen` keys. Do not put secrets or sensitive personal data in `metadata`.

The per-file base64 limit is 28,000,000 characters (about 20 MB decoded), within a 50 MB JSON request limit. There is no separate attachment-count limit. `delivery: "inline"` sends content to the model; `"context"` exposes a manifest to the model and keeps the bytes available to tools. When omitted, runtime delivery policy applies.

Sessions

Method	Path	Purpose
`POST`	`/sessions`	Prepare a session before its first message.
`GET`	`/sessions/history`	List sessions, up to 500; accepts `?scope_id=`.
`GET`	`/sessions/history/:id`	Read a session's history.
`PATCH`	`/sessions/history/:id`	Update session metadata.
`DELETE`	`/sessions/history/:id`	Delete a session.
`GET`	`/sessions/pending-approvals`	List sessions waiting for an operator answer.

On a hosted runtime, these routes require authentication. History and pending-approval lists hide sessions owned by another verified user and, for a token with a `scope_id` claim, sessions in other scopes. Reading, updating, or deleting a hidden session returns `404`, the same response as a missing session. Updates and deletions check both live and persisted ownership. Sessions without an owner remain shared, and trusted server credentials without a verified user ID retain access to owned sessions.

For a token without a `scope_id` claim, the `?scope_id=` query is a caller-supplied list filter, not an ownership credential. Lists filter ownership after fetching up to 500 rows, so fewer visible sessions can be returned even when other accessible sessions exist. For tenant-specific rules or calls using server credentials, have your application backend authorize each operation and retain the runtime credential.

Interactions

Method	Path	Purpose
`POST`	`/chat/sessions/:id/ask-user-response`	Answer an ordinary chat confirmation with `{correlation_id, approved}`. `ask_id` is also accepted.
`POST`	`/chat/sessions/:id/messages`	Queue `{message}` for the agent while a turn runs. Returns `202 {status, message_id}`, `409 {error: 'no_active_turn'}` when no turn is accepting messages, or `422 {error: {code: 'MESSAGE_BLOCKED'}}` when a preInput guardrail blocks the message.
`GET`	`/chat/sessions/:id/turns/:turnId`	Observe an active or buffered resumable turn over SSE (`?after=<seq>`).
`POST`	`/chat/sessions/:id/turns/:turnId/cancel`	Cancel an active turn.
`POST`	`/api/sessions/:id/answer`	Answer a parked durable workflow.
`POST`	`/api/sessions/:id/cancel`	Cancel a durable workflow session.
`POST`	`/api/sessions/:id/restart`	Run a durable tool session's tool again in a new session.

Use the `correlation_id` from `confirmation_required` for an ordinary confirmation. The chat loop waits for that response. Choice selections from `ask_choice` are sent as the next chat message.

A `durable_approval_required` event identifies a separate parked session. Answer that session through the durable workflow API; see Agent Workflows.

System and artifacts

Method	Path	Purpose
`GET`	`/health`	Health check (`{ status: "ok" | "draining" | "starting", version, uptime_ms, active_sessions }`).
`GET`	`/api/events`	Runtime state subscription, when mounted by the host.
`GET`	`/api/files/:fileId`	Download a file published by `publish_file` in a sandbox session.
`GET`	`/api/sessions/:sessionId/artifacts`	List structured artifacts registered in the session.
`GET`	`/api/sessions/:sessionId/artifacts/:eventId/content`	Download a verified artifact's bytes.
`POST`	`/channels/:channelType/webhook`	Inbound messaging-channel webhook.

SSE event types

Chat frames contain a JSON object with `type` , `timestamp` , and event-specific fields. The server sends `data:` frames without a separate `event:` line.

Common events are listed below. The React SDK exports `SSEEvent` from `@amodalai/react` for typed chat stream handling.

Type	Fields	Purpose
`init`	`session_id`	Session created or resumed
`text_delta`	`content`	Incremental answer text
`thinking_delta`	`content`	Model reasoning output, when supplied
`tool_call_start`	`tool_name` , `tool_id` , `parameters` , `running_label?` , `completed_label?`	Tool execution started
`tool_label_update`	`tool_id` , `running_label?` , `completed_label?`	Tool display label changed
`tool_call_result`	`tool_id` , `status` , `result?` , `content?` , `parameters?` , `duration_ms?` , `error?`	Tool result or error

Type	Fields	Purpose
`tool_log`	`tool_name`, `message`	Handler progress log
`subagent_event`	`parent_tool_id`, `agent_name`, `event_type`, `tool_name?`, `tool_args?`, `result?`, `text?`, `error?`	Specialist activity
`skill_activated`	`skill_name`	Skill loaded
`file`	`id`, `filename`, `mime_type`, `size_bytes`, `url`, `artifact?`	Sandbox file or registered artifact
`widget`	`widget_type`, `data`	Structured content for the client
`confirmation_required`	`endpoint`, `method`, `reason`, `escalated`, `params?`, `connection_name?`, `correlation_id?`	Ordinary tool approval
`approved` / `denied`	`resource_type`, `preview_id`	Approval decision applied
`ask_choice`	`ask_id`, `question`, `options`, `multi?`	Question with selectable options
`durable_approval_required`	`session_id`, `tool_name`, `question?`, `options?`, `allow_edits?`	Parked workflow approval
`agent_handoff`	`from`, `to`, `removed_tools`	Session changed its root agent
`user_message_delivered`	`message_id`, `text`	Queued mid-turn message delivered
`input_redacted`	`text`	User input was redacted
`compaction_start`	`estimated_tokens`, `threshold`	Context summarization started
`compaction_end`	`tokens_before`, `tokens_after`, `compaction_tokens`	Context summarization completed
`warning`	`message`	Non-fatal problem
`error`	`message`	Execution error
`done`	`reason?`, `turns?`, `trace_id?`, `usage?`	Turn completed

Streaming example

```
data: {"type":"init","session_id":"session-123","timestamp":"2026-09-07T10:00:00Z"}

data: {"type":"text_delta","content":"Hello.","timestamp":"2026-09-07T10:00:01Z"}

data: {"type":"done","reason":"model_stop","turns":1,"usage":
{"input_tokens":100,"output_tokens":2,"cached_tokens":0,"total_tokens":102},"timestamp":"2026-09-
07T10:00:01Z"}
```

`trace_id` is the trace of the run that produced the answer, present when the runtime exports traces. The chat widget sends it with a thumbs up or down, so the feedback links to that run in Amodal.

Done reasons

Reason	Meaning
`model_stop`	The model finished without requesting more tools, or a tool ended execution.
`max_turns`	The run reached its model-turn limit, 50 by default.
`user_abort`	The request was cancelled or the client disconnected.
`error`	An unrecoverable error ended the run.
`budget_exceeded`	Accumulated session tokens reached `max_session_tokens`. The check runs before the run starts and between steps, so one model call can exceed the budget.

When a top-level run stops on `max_turns` or `budget_exceeded`, one final model call streams a summary as ordinary text, and the `done` event keeps the original reason. For `max_turns` the summary says what was done, what is left, and what the user should say to continue. For `budget_exceeded` it says what was done and what is left, and that a new conversation is needed. A tool call requested in that final turn is not run. If the summary call fails or writes no text, the runtime streams a fixed message naming the limit. A run that starts at or over the budget makes no model call. Cancelled runs, subagents, and runs with a turn limit of 1 make no summary call.

Normal completion emits `done` with usage and the reason. A disconnected client cannot rely on receiving it; read the persisted session history to inspect an interrupted request.

Building a client

`streamChat` parses the chat stream and accepts an abort signal and runtime token. This helper requires the React package and its peer dependencies; it can be used without rendering a chat component.

```
import { streamChat } from '@amodalai/react';

async function chat(runtimeUrl: string, token: string, message: string, sessionId?: string) {
  for await (const event of streamChat(
    runtimeUrl,
    {
      message,
      ...(sessionId ? { session_id: sessionId } : {}),
    },
    undefined,
    token,
  )) {
    if (event.type === 'init') sessionId = event.session_id;
    if (event.type === 'text_delta') console.log(event.content);
    if (event.type === 'error') throw new Error(event.message);
  }
  return sessionId;
}
```

Store the returned session ID and pass it with the next message. This minimal example prints text; it does not render widgets or answer confirmations. Use `useChat` or `AmodalChat` for those interactions.

Session lifecycle

A request without `session_id` creates a session. Later requests with that ID resume the conversation. The runtime records the verified user ID of the session's creator and refuses to reuse the session for a request whose verified user ID differs, starting a fresh session instead. A caller with no verified user ID, such as an API key or a service token, can resume any session. When the caller's token has a `scope_id` claim, the runtime also refuses to resume a session stored in another scope and starts a fresh session instead. Without that claim, the runtime does not compare the session's scope with the request. Your application backend must authorize a resume request before forwarding it.

`POST /sessions` creates a session ahead of the first message. It accepts the chat body's `agent`, `skill`, `deploy_id`, `max_session_tokens`, `model`, `scope_id`, and `context`, builds the session from them, and returns `201` with `{session_id, idle_ttl_ms}`, where `idle_ttl_ms` is the runtime's idle session timeout in milliseconds. The first message that carries that ID skips the session setup and goes straight to the model. Send the same fields you will chat with, because a live session keeps its skill, deploy, model, and token budget. A prepared session is not persisted until its first turn, and an unused one expires with the idle timeout. The React SDK's chat hooks prepare a session when they mount; see Chat hooks.

The runtime evicts a session from memory after 30 minutes idle. Hosted sessions persist in the platform, so eviction does not delete history and the next request reloads the conversation. Local development keeps sessions and stores in memory only, so they are lost on restart.

An ordinary chat stream disconnect cancels the active request. Sending the same `session_id` continues the conversation; it does not replay the interrupted SSE response. Use durable workflows when work must continue

independently of the connection.

Context compaction

When estimated context passes 90% of the model's context window, the runtime manages history in two ways:

- **Tool-result clearing.** Once the conversation holds more than 15 tool-result messages, results older than the latest 5 become short summaries or cleared-result markers. Selection is by recency, not size.
- **Compaction.** The runtime summarizes older turns with the session's model and keeps the latest 6 user turns verbatim. The summary is capped at 4,000 tokens and covers the current task, key data, actions taken, errors, and next steps. It emits `compaction_start` and `compaction_end`.

Exact wording and tool-result details can be lost. Save facts that must survive in a store. If summarization fails, the run continues with the existing history; after 3 consecutive failures, compaction stops for that run. These thresholds are not configurable from agent source.

REFERENCE

Runtime Event Bus

The runtime publishes session changes, store writes, bundle reloads, messaging-channel activity, and tool-call activity through `GET /api/events`. Use these events to refresh data in your application.

Chat responses use a separate POST stream with `text_delta`, `tool_call_start`, and `done` messages. See Runtime Server. The event bus reports state changes across sessions; it does not replace authenticated data queries.

Subscribe

This browser example uses the current origin and its hosted session. A cross-origin or bearer-authenticated client needs an appropriate authenticated connection; native `EventSource` cannot set an Authorization header.

```
const events = new EventSource('/api/events');
events.addEventListener('session_created', (message: MessageEvent<string>) => {
  const event = JSON.parse(message.data);
  console.log('Session created:', event.sessionId);
});
```

Each frame carries an `event:` line with the event type, so register a named listener for each type the view uses. An `onmessage` handler receives nothing.

Call `events.close()` when the view is disposed. The server sends a comment heartbeat every 15 seconds. `amodal dev` and hosted deployments both serve this route.

A hosted event stream carries every session's events. Operator credentials, such as the agent's API key, Studio, or a runtime token without a `scope_id` claim or the `user` role, receive full events. Every other subscriber, including the hosted browser session and the React SDK's `useAgentEvents`, receives each event without its content fields: `title`, `message`, `error`, `channelUserId`, `messagePreview`, `replyPreview`, `argsSummary`, and `targetUrl`. `amodal dev` sends full events to every subscriber.

Reconnect-and-resume

`EventSource` saves the SSE `id` and sends it as `Last-Event-ID` on reconnect. Treat this ID as an opaque cursor:

- `amodal dev` uses a process-local sequence and retains the latest 200 events. Its cursor and buffer reset when the process restarts.
- Hosted deployments routed through the event service use Redis stream IDs, with retention configured by the platform. Do not substitute the event payload's `seq` for this cursor.

The server replays retained events after the cursor, then streams live events. Missing events outside retention cannot be recovered from this subscription. Refresh the underlying data after a reconnect if your view needs a complete state. A gap in numeric `seq` can also come from filtered event types, so it is not proof of data loss.

Event catalog

Session events

Type	Fields	When
`session_created`	`sessionId`, `agentId`	A new session was created (chat, admin, automation).
`session_updated`	`sessionId`, `agentId`, `title?`, `status?`, `answeredBy?`	A session was persisted after a message, title change, or metadata update.
`session_deleted`	`sessionId`	A session was deleted via the API.

Store events

Type	Fields	When
`store_updated`	`storeName`, `operation`, `count?`	A document was written or deleted. `operation` is `"put"` or `"delete"`. A purge of expired documents emits `"delete"` with `count` set to the number purged, and `storeName` is `"*"` when the purge covers every store.

Config events

Type	Fields	When
`manifest_changed`	None	The agent manifest (connections, skills, automations) was reloaded after a source or deploy change.
`reload_failed`	`message`	A local development bundle reload failed; the server keeps serving the previous bundle.

Channel events

Type	When
`channel_message_received`	An inbound message arrived on a messaging channel.
`channel_reply_sent`	The agent's reply was delivered back to the channel.
`channel_session_created`	A channel conversation started a new session.
`channel_auth_rejected`	A channel webhook failed authentication.

Tool-call events

`tool_call_started`, `tool_call_completed`, and `tool_call_failed` report execution activity. `amodal dev` excludes them by default; add `?include=tool_calls` or individual type names to include them. Hosted routes ignore `include` and send every event type.

Types

All events carry `seq`, `timestamp`, and `type` fields plus the event-specific payload. The React SDK exports `AgentEvent`, which accepts runtime event fields as unknown values. Narrow those fields before using them:

```
import type { AgentEvent } from '@amodalai/react';

function handleEvent(event: AgentEvent) {
  if (event.type === 'session_updated' && typeof event['sessionId'] === 'string') {
    console.log(event['sessionId'], event['status']);
  }
}
```

Events and data access

Use events to know when to refresh a session list, automation status, or store-backed view, then fetch the current data through the corresponding HTTP API. Receiving an event is not permission to read the referenced session or store.