Atlas Development Guide

---

type: guide title: Atlas Development Guide status: Draft status_detail: "DG1 S3 — front-door doc for building on Atlas, proven against a real worked-example plugin (dev-guide-proof); author-from-guide gaps folded back. Map, not rewrite." author: devops-lead drafted: 2026-06-09 updated: 2026-06-09 summary: "The one doc a Codex or Claude developer reads before touching Atlas: how the dev loop and runtimes work, where plugins/skills/code live, how to use the SDK, how to write tests, and the documentation and delivery standards to follow." source: "Workspace SoT — knowledge/atlas/development-guide.md. The BookStack copy on docs.netos.io is a rendered mirror, not a second source of truth." related:

- knowledge/atlas/capability-spine/dev.yaml
- team/devops-lead/outputs/atlas-operating-foundation-design.md
- team/devops-lead/outputs/atlas-netos-delivery-master-plan.md

---

Atlas Development Guide

This is the single page you read before you build on Atlas. It is a **map**, not an encyclopedia: it tells you where everything lives and how the pieces fit, and it points you at the authoritative doc for each piece rather than repeating it. When this guide and a linked design doc disagree, the design doc wins for its own subject and this guide should be corrected.

How capabilities are cited. Atlas describes itself through the **Capability Spine** — a grouped index of every Atlas capability, each citable by a stable id of the form `group.slug`. Throughout this guide, a capability is cited by that id and linked to the live Capabilities index. The ids are stable, so cite them freely; to jump to one, open the index and filter by the id (the page has a filter box that matches on id, name, surface, and owner). The spine itself is `gates.capability-spine`.

> **Source of truth.** This file — `knowledge/atlas/development-guide.md` in the workspace — is authoritative. It renders in Atlas at `/file?path=knowledge%2Fatlas%2Fdevelopment-guide.md`. The copy published to BookStack (docs.netos.io) is a rendered mirror for the wiki audience; never edit the wiki and expect it to flow back.

---

1. What Atlas is, for a developer

Atlas v2 is an agent platform assembled from a small kernel plus plugins, driven by LLM runtimes, deployed by Ansible-pull from Git. As a developer you work across five repositories, all cloned under `netos-gitlab/netos-agents/` in this workspace and originating on `uks-git01.prod.netos.io`:

Repo (clone path under `netos-gitlab/netos-agents/`)	Owns
`netos-atlas`	The **kernel** (`packages/kernel/`), the web app (`apps/web/`, Next.js), and the API (`apps/api/`, Fastify). Kernel modules live in `modules/` (`files`, `playbook-runtime`, `scheduler`).
`netos-atlas-plugins`	Every **plugin** under `plugins/<id>/`. This is where most feature code lands.
`netos-atlas-sdk`	The published TypeScript SDK, `@netos/atlas-sdk` (`packages/ts/`). Plugins import their types and helpers from here.
`netos-atlas-deploy`	The **deploy** repo (Ansible). Symlinked into the workspace at `platforms/ansible/playbooks/atlas-deploy/`. Stages workspace files onto hosts, fetches plugin source, builds, flips the release.
`netos-mcp`	The MCP server exposing the `netbox.*` / `support.*` data tools and the offline `bin/dryrun-playbook` harness.

The **kernel / plugin / SDK boundary** is the thing to internalise first:

- The **kernel** owns the runtime, the database, auth, the route table, and the module lifecycle. You rarely change it.
- A **plugin** is a self-contained unit that registers routes, actions, jobs, and UI against a `PluginContext` the kernel hands it at `init()`. This is your primary build surface.
- The **SDK** is the typed contract between the two: a plugin imports `PluginContext` and the manifest helpers from `@netos/atlas-sdk` and never reaches into kernel internals.

For the why behind this shape — capability as the primary object, the in-Atlas dev loop, the authoring kit — read the Atlas Operating Foundation design (section 8 is the dev loop, section 6 the authoring kit). This guide does not restate it.

---

2. The dev loop — author → test → publish → deploy

This is the one section that is genuinely new connective tissue; everything else links out. The loop has four legs, and a change is not done until it has been round all four.

```
flowchart LR
  A["Author<br/>(Codex / Claude runtime)"] --> B["Test<br/>(colocated suites + CI)"]
  B --> C["Publish<br/>(commit → main → push to uks-git01)"]
  C --> D["Deploy<br/>(atlas-deploy → host-verify)"]
  D -->|gap found| A
  classDef leg fill:#1f2937,color:#e5e7eb,stroke:#374151;
  class A,B,C,D leg;
```

Author. Code is written by one of two dev runtimes. The `dev.codex-runtime` runs Codex on lab02 `atlas-dev` via a ChatGPT account login over HTTP dispatch; the `dev.claude-runtime` is the second authoring runtime, driven through the model router. Both reach their backend models through `dev.model-router`, the local proxy that selects bound models behind one endpoint (design: Goose runtime & model-router). Which runtime, model, credential, and bot identity a given task uses is selected by a **Runtime Profile** — see `dev.runtime-profiles` and the Development Agency gate-chain plan. The end-to-end intent → runtime routing is described in the dev-loop intent/runtime routing design, and the in-Atlas loop overall in the in-Atlas dev-loop design. The runtimes, profiles, and the per-host/role bots that drive them run on the `dev.dev-lab-fleet` (`eus-az2-atlas-*`), with lab01 as the build/control host and lab02 as the active dev/UAT surface; see the internal dev-lab strategy design and the dev-lab fleet host-vars design.

Test. Write tests *with* the code (section 6). CI runs them on every change and fails red on a real failure.

Publish. Commit to `main`. Workspace files (KB, spine YAML, skills) sync via `source.workspace-mirror`; repo clones push to uks-git01 via `source.gitlab-sync` (the `git-repo` wrapper). The deploy repo is its own clone, `source.atlas-deploy-repo`, and is **not** workspace-mirrored — it must be pushed explicitly.

Deploy. Run the gate's named `atlas-deploy` command against the target host, then verify ground truth on the host — never self-report (section 7). For multi- step gated work the `gates.gate-runner` drives the steps and `gates.gated-delivery` is the process they follow.

---

3. Where code lives

Four homes, by kind of thing:

- **Plugins** → `netos-atlas-plugins/plugins/<id>/`. A plugin is a directory with a `plugin.json` manifest, a `src/` (TypeScript, `entrypoint` is usually `src/index.ts`), colocated `tests/`, and optionally `schema/` and a `settings.schema.json`. The canonical read-only template to clone is `plugins/system-capability-spine/` (itself a clone of `system-designs-index`).
- **Agent skills** → `knowledge/skills/<skill>/`. These are *behaviours* an agent performs (e.g. `code_edit`, `host_deploy`, `branch_push`), deployed to hosts via the skill-bundle mechanism, not compiled into a plugin.
- **Kernel modules** → `netos-atlas/modules/<module>/`. Core runtime concerns (`files`, `playbook-runtime`, `scheduler`). You touch these only when a change is genuinely kernel-level.
- **Workspace knowledge** → `knowledge/` (this guide, the capability spine YAML, playbooks, brand hub). Authoritative reference; staged onto hosts by the deploy.

The "two kinds of skills" distinction

The word "skill" names two different layers; keep them separate (Operating Foundation section 4.4):

- An **agent skill** is *how* an agent does something — a behaviour under `knowledge/skills/`.
- An **MCP tool / capability** is *what* data or action exists — the `netbox.*` / `support.*` surface served by `netos-mcp` (`mcp.netos-mcp-server`).
- A skill **uses** tools. State the link explicitly when you author one (e.g. `kb-answer → support.find_topic / support.read_kb`).

---

4. Using the SDK

Plugins build against `@netos/atlas-sdk` (the FND1 deliverable that consolidated 59 local type mirrors down to one published package — the story is in the SDK mirror re-audit, and consumer docs are in the SDK README).

Import types — never re-declare them locally. The SDK is **types + helpers, not a manifest builder**: there is no `defineManifest`/`definePlugin` export. The manifest is hand-authored JSON (`plugin.json`, below); from the SDK you import the runtime contract your `init()` builds against:

```
import type { PluginContext, ModuleInitResult } from '@netos/atlas-sdk';
```

`PluginContext` is the single object the kernel hands your `init(ctx)`. Its surface (from `packages/ts/src/index.ts`):

Field	What you use it for

`ctx.routes`	Register HTTP routes (`ctx.routes.get/post(...)`), with a `requirePerm` guard.
`ctx.actions`	Expose and call cross-plugin actions (the `provides_api` surface).
`ctx.settings` / `ctx.secrets`	Read plugin config and secrets. **Keys must be lowercase** — the seeder lowercases env keys before schema match, so an upper-case key seeds as `unknown_key`.
`ctx.db`	Scoped SQL handles, but only for the bindings you opted into via `requires.db` (see allowlist below).
`ctx.audit` / `ctx.capture`	Emit audit records; resolve capture level.
`ctx.health`	Register a health probe (surfaced on the host health endpoint).
`ctx.flow`	Ambient flow-frame helpers for orchestration-driven plugins.
`ctx.log`	Structured logging.
`ctx.api`	The stable cross-plugin read API surface.

The manifest (`plugin.json`)

Authored at `plugins/<id>/plugin.json`. The **enforced** gate is the kernel's zod `ManifestSchema` (`netos-atlas/packages/kernel/src/manifest.ts`) — the module loader runs `ManifestSchema.parse()` on it before the plugin is loaded, and the schema is `.strict()`, so an unknown top-level field or an out-of-enum `category` rejects the whole manifest and the plugin silently never loads. The companion JSON Schema is `netos-atlas-sdk/schemas/plugin.schema.json` (in the **SDK** repo, not the plugins repo), hand-kept in lockstep with the zod schema. The fields that matter most:

- `id`, `name`, `version`, `author`, `language: "ts"`, `entrypoint: "src/index.ts"`.
- `pack` — which deploy pack the plugin belongs to. This field is the **source of truth** for pack membership; the deploy catalog mirrors it. **A plugin only loads on a host if its slug is in an enabled pack.** The catalog lives under the `atlas_pack_catalog:` key in `netos-atlas-deploy/group_vars/all/packs.yml`, so a new `core` plugin must be added to the `atlas_pack_catalog.core` list there (mirroring its `pack: core`) or it is filtered out at install and 404s after deploy.
- `requires.plugins` — other plugins this one depends on.
- `requires.db` — the **DB-binding allowlist**. A scoped `ctx.db` handle appears *only* if the key is in `requires.db` **and** registered in the kernel's `db-handles.ts` (the currently registered keys are `usageRollups`, `mcpRequests`, `mcpCalls` — camelCase); an unknown key is dropped at bind time (not rejected at parse), so it degrades silently with a misleading warning. Declare `[]` if you need no DB.
- `provides_api` — the action surface other plugins may call.
- `routes`, `jobs`, `ui` — declared surfaces (the `ui` entries become Atlas pages).

For the worked manifest, read `plugins/system-capability-spine/plugin.json` — a read-only, `requires.db: []`, single-UI-page plugin that is the cleanest minimal example.

---

5. Writing a plugin

The reliable path is **scaffold from a proven read-only plugin, then adapt**:

1. **Clone the template.** Copy `plugins/system-capability-spine/` to `plugins/<your-id>/`. Rename `id`, `name`, routes, and `provides_api` in `plugin.json`. Set `pack` to the pack you intend (`core` for always-on).
2. **Write `src/index.ts`.** Export `async function init(ctx: PluginContext)`. Register routes with an explicit `requirePerm` (e.g. `admin.plugins.read`), and return any `actions` you expose. Keep the handler thin; put logic in sibling modules (`src/scan.ts`, etc.).
3. **Settings & secrets.** Declare a `settings.schema.json` and read values via `ctx.settings.get(...)` / `ctx.secrets`. Lowercase every key.
4. **Pack allowlist.** If the plugin is new, add its id under the matching `atlas_pack_catalog.<pack>` list in `netos-atlas-deploy/group_vars/all/packs.yml` (mirroring its `plugin.json` `pack`). This is a deploy-repo edit and must be pushed to uks-git01 (`source.atlas-deploy-repo`).
5. **Provide an API only if another plugin needs it.** `provides_api` plus `ctx.actions` is how plugins call each other; don't expose internals you don't have to.

The DG1 **worked example** — the `dev-guide-proof` plugin authored solely by following this guide — is the concrete "now you do it" exhibit; see section 9.

---

6. Writing tests

Tests are real and gating now (the FND2 gate made CI execute the colocated suites and fail on red). Two patterns coexist:

- **Colocated `*.test.ts` next to `src/`**, compiled and run via the plugin's own `pnpm test`. The plugins-repo CI `test` stage (`scripts/run-plugin-tests.mjs`) runs every plugin's suite on every change; a single failing test fails the pipeline. `netos-atlas` has the equivalent job for its kernel suites.
- **`node:test` runner suites** under `tests/` (e.g. `tests/*.test.mjs`) for plugins that test against fixtures rather than the type build.

Local checks before you push:

- **Typecheck** — the repo-root eslint config has no TS parser, so per-file lint errors are noise. Use `tsc` by tsconfig instead: `node node_modules/typescript/bin/tsc -p <abs path to plugin tsconfig>`.
- **Unit tests** — run the plugin's `pnpm test` in its own directory.
- **Playbook dry-run** — for playbook/journey changes, validate offline against fixtures with `netos-mcp/bin/dryrun-playbook` before any host run. The dry-run / approval discipline is `safety.dry-run-approval`.

CI is the backstop, not the substitute: a green local run plus a green pipeline on uks-git01 is the bar before deploy.

---

7. Publishing & deploying

Publish

- **Workspace files** (KB, `knowledge/atlas/capability-spine/*.yaml`, this guide, `knowledge/skills/`) are committed to workspace `main`; `source.workspace-mirror` syncs them. No feature branches that outlive their gate.
- **Repo clones** (`netos-atlas`, `netos-atlas-plugins`, `netos-atlas-sdk`) push to uks-git01 with the `git-repo` wrapper (`source.gitlab-sync`), which injects the right token. Pushes to uks-git01 need `GIT_SSL_CAINFO=/usr/lib/python3/dist-packages/certifi/cacert.pem`.
- **The deploy repo** (`netos-atlas-deploy`) is its own clone and is **not** workspace-mirrored — edits to deploy tasks must be committed and pushed separately (`source.atlas-deploy-repo`).

Deploy

- **Full deploy** stages workspace files onto the host mount, fetches `plugins-src@main`, rebuilds web, and flips the release: `platforms/ansible/bin/ansible deploy atlas-deploy --limit eus-az2-atlas-lab02`.
- **Plugins-only fast deploy** (`atlas-plugins.yml`) re-fetches plugin source and restarts (~3 min vs ~15) — the right path when only plugin code changed and no new workspace file needs staging.
- **No migrations run automatically.** `atlas-deploy` applies neither kernel nor plugin DB migrations; a schema change must be applied and verified on the host by hand.
- **Staging gaps are real.** The deploy only stages the files its tasks name. A new workspace doc (like this guide) needs its own staging task, or it 404s on the host even though the commit landed.

Verify on the host — never self-report

A deploy step is "done" only when ground truth on the host confirms it (`safety.fail-closed-verify`): read the live release with `readlink current`, confirm the plugin loaded, check the journal, and probe the real route with a host-minted admin cookie. Build-then-flip means a broken redeploy leaves the prior release `current`, so a failed verify is safe to retry. Gate verify scripts must be fail-closed and must avoid `echo | grep -q` under `pipefail` (SIGPIPE races a false-negative pause); use here-strings.

---

8. Documentation & delivery standards

- **Design docs.** Anything that describes how we'll build X before building it goes to `team/<role>/outputs/<topic>-design.md` with `type: design` YAML frontmatter (status Draft → Approved → Implemented → Deprecated). The filename `-design.md` heuristic puts it in the `gates.designs-index` (`/designs`). Use ```` ```mermaid ```` blocks, not ASCII art; if you `classDef` a light `fill:`, also set a dark `color:` or the text renders unreadable.
- **The capability spine.** When you ship a capability, add or update its entry in `knowledge/atlas/capability-spine/<group>.yaml` and let the gap-ledger loop keep coverage honest. Entries are validated (schema, unique id, `group.slug` prefix, non-empty `docs`) by the spine plugin's `bin/validate-spine.mjs`. The `docs.kind` enum is `bookstack | design | schema | tool | playbook | skill | readme` — there is no `guide` kind, so reference markdown uses `readme`. Adding an entry needs no kernel deploy: commit the YAML, deploy stages it, a plugin **Refresh** re-scans.
- **Doc standard.** The house Atlas docs standard is `knowledge.docs-overhaul`; playbooks are indexed under `knowledge.playbook-index`.
- **Voice.** Internal docs (this guide, designs, handovers) use plain US-English and are unrestricted on punctuation. The em-dash ban and the brand voice apply only to customer-facing copy — that hub is `knowledge.brand-voice-hub`.
- **Deploy on close.** A gate or milestone is not closed until its changes are deployed to the surface where they run, with the deploy command's output as evidence. Exceptions must be named explicitly in the design doc.
- **Gated delivery.** Multi-step builds run as numbered gates with a handover doc per gate (`gates.gated-delivery`, full process in the gated-delivery process doc). The `gates.gate-runner` executes the steps; keep close steps small so they don't overrun the wall-clock budget. Models a task runs under are selected by `models.profiles`.
- **Sandbox & security.** Untrusted file input flows through `sandbox.file-ingest`; code execution is confined by `sandbox.nsjail` (use a *finite* `rlimit_fsize`, never `max`/inf, or writes fail with EIO). The operational baseline is `SECURITY.md`.

9. Worked example — `dev-guide-proof`

The proof that this guide is sufficient is a minimal plugin authored **solely by following it**: `netos-atlas-plugins/plugins/dev-guide-proof/`. It is a read-only TypeScript plugin scaffolded from `system-capability-spine`, with:

- `plugin.json` — `id: dev-guide-proof`, `pack: core`, `language: ts`, `requires.db: []`, `requires.plugins: []`.
- `src/index.ts` — one GET route, `/api/plugins/dev-guide-proof/ping` → `{ ok: true, guide: "dev.development-guide" }`.
- a colocated test.

It exercises the highest-value path the guide documents: importing `PluginContext` / `ModuleInitResult` from `@netos/atlas-sdk` with no local mirror, hand-authoring a strict-schema-valid manifest, adding the slug to `atlas_pack_catalog.core` in `packs.yml`, and running a colocated test. It is additive and disposable — it can stay as a permanent reference exhibit or be removed with no schema or DB impact.

Validated (DG1 S3), all from inside the plugin dir:

- `pnpm install --no-frozen-lockfile` — links the SDK via the `file:` dep.
- `node node_modules/typescript/bin/tsc -p tsconfig.json --noEmit` — typecheck, exit 0.
- `node --test --import tsx './tests/**/*.test.mjs'` — colocated suite, 2/2 pass.
- Manifest load: `plugin.json` parses clean against the kernel `ManifestSchema` (the exact gate `module-loader.loadOne()` runs before a plugin reaches `list()`).

Gaps found while authoring solely from this guide, folded back above: the SDK has no `defineManifest` helper (section 4 import corrected); the manifest JSON Schema lives in the SDK repo and the enforced gate is the kernel's strict zod `ManifestSchema`, not a `schemas/plugin.schema.json` in the plugins repo (section 4); and the pack catalog is nested under `atlas_pack_catalog:` with the plugin.json `pack` field as source of truth (sections 4 and 5). The guide and the real path are now in sync.

Capabilities cited in this guide

Each links to the Capabilities index; filter by the id to open the entry.

- `dev.codex-runtime`, `dev.claude-runtime`, `dev.model-router`, `dev.runtime-profiles`, `dev.dev-lab-fleet` — the dev runtimes, routing, profiles, and fleet.
- `gates.gate-runner`, `gates.gated-delivery`, `gates.designs-index`, `gates.capability-spine` — delivery.
- `source.gitlab-sync`, `source.workspace-mirror`, `source.atlas-deploy-repo` — publish & deploy plumbing.
- `sandbox.file-ingest`, `sandbox.nsjail`, `safety.fail-closed-verify`, `safety.dry-run-approval` — sandbox & safety.
- `models.profiles`, `mcp.netos-mcp-server`, `knowledge.docs-overhaul`, `knowledge.playbook-index`, `knowledge.brand-voice-hub` — models, data, docs.

Created 2026-06-09 10:02:44 UTC by Netos AI Agent
Updated 2026-06-09 10:37:20 UTC by Netos AI Agent