Install AIP

For most users, the supported entry point is the public `getaip` npm package. It installs the signed GetAIP 2.1.0 native distribution and can connect a supported agent to the verified GetAIP MCP server. Developers and operators can still build the same CLI, server, and connector-fleet components from a pinned source revision when they need source-level review or a custom artifact.

DOCUMENT TYPE	how-to	AUDIENCE	developer
APPLIES TO	GetAIP 2.1.x / AIP 1.0	REVIEWED	e6222fc36ebd7d8a723d773606c073d5da9cd281

The npm package is a verified setup launcher, not a general JavaScript library. The command shown by npm, `npm i getaip`, only adds the launcher to the current project; it does not run setup or connect an agent.

Install the signed release with npm

You need:

- Node.js `22.14.0` or newer with npm and `npx`;
- macOS or Linux on ARM64 or x64;
- network access to the npm registry and the GetAIP GitHub Release assets;
- write access to the current user's platform-native data and configuration directories.

Install or repair the signed native distribution and create the default loopback configuration:

```
npx getaip setup
```

This command does not configure an agent unless you also select a client and scope. To reproduce the currently reviewed release exactly, pin the package version:

```
npx getaip@2.1.0 setup
```

Do not prefix setup with `sudo`. The installer deliberately writes to user-owned platform directories and retains the signed manifest, signature, artifact digests, active version, and rollback state for later verification.

Connect a supported agent

Choose one command for a user-global configuration:

Agent	Command
Codex	`npx getaip setup --codex --global`
Claude Code	`npx getaip setup --claude --global`
Cursor	`npx getaip setup --cursor --global`
Gemini CLI	`npx getaip setup --gemini --global`
OpenCode	`npx getaip setup --opencode --global`

You can select multiple clients in one command, for example:

```
npx getaip setup --codex --claude --global
```

For a project-local configuration, run setup from the project root and use `--project` instead:

```
npx getaip setup --codex --project
```

Use `--project-root /absolute/path/to/project` when the target is not the current directory. Client configuration requires exactly one scope: `--global` or `--project`.

Preview and verify setup

Inspect the signed plan and proposed client mutation without changing the filesystem:

```
npx getaip setup --codex --global --dry-run
```

For automation, request the stable JSON output:

```
npx getaip setup --codex --global --dry-run --output json
```

The npm launcher accepts only the `setup` command. It selects a compatible macOS or Linux bootstrap, verifies the Ed25519-signed release manifest against its embedded trust store, enforces bounded downloads and approved GitHub release redirects, verifies artifact size and SHA-256 identity, and only then executes native setup. A successful non-dry-run reports the active version and every configured client adapter.

Build from source

Use the source path for implementation development, reviewed custom builds, connector-fleet assembly, or qualification work. The Rust workspace packages remain non-publishable on crates.io, so this procedure starts from a source checkout. A successful build proves that the selected source compiles in the current environment; it does not by itself qualify the resulting artifact for production.

Choose an installation target

Select the smallest target that matches the work you intend to perform:

Target	Build	Use it for
Local native AIP	`getaip-server` and `getaip`	Run a product-neutral daemon and make native or compatibility-profile calls
Connector fleet	`getaip-server`, `getaip`, `aip-connector-control-plane`, and selected standalone hosts	Keep product adapters and credentials outside the central daemon
Legacy migration	`getaip-server-legacy-bundled`	Move an existing bundled deployment to separate connector processes
Rust integration	`aip` facade with selected features	Embed AIP types and product-neutral implementation components in Rust code

The `getaip-server` dependency graph contains no product connector crate. Do not use `getaip-server-legacy-bundled` for a new deployment; it retains only the older bundled composition needed during migration.

Prerequisites

You need:

- a checkout containing commit `e6222fc36ebd7d8a723d773606c073d5da9cd281`;
- Rust `1.88` or newer with Cargo, `rustfmt`, and Clippy;
- write access to a dedicated Cargo target directory;
- enough local disk space for Rust dependencies and release artifacts.

`rust-toolchain.toml` selects the moving `stable` channel, while `Cargo.toml` declares Rust `1.88` as the minimum supported version. Repository CI at this revision runs the full Linux jobs on Ubuntu 24.04 and a portable-core subset on macOS 15. It has no Windows job, so use a validated Linux environment for release-affecting work that would otherwise run on Windows.

Check the local toolchain before building:

```
rustc --version
cargo --version
```

Both commands must exit successfully, and `rustc` must report version `1.88.0` or newer.

Pin the source revision

Run these commands from the repository root. Inspect local changes before switching revisions:

```
git status --short
git rev-parse --verify e6222fc36ebd7d8a723d773606c073d5da9cd281^{commit}
```

If `git status --short` reports work you need, preserve it before continuing. Then select the reviewed source in detached-HEAD mode:

```
export AIP_SOURCE_REVISION=e6222fc36ebd7d8a723d773606c073d5da9cd281
git checkout --detach "$AIP_SOURCE_REVISION"
test "$(git rev-parse HEAD)" = "$AIP_SOURCE_REVISION"
test -z "$(git status --porcelain)"
```

The last two commands must exit with status `0`. A dirty checkout produces a development artifact whose identity is not the recorded commit alone.

Build the core tools

Build the product-neutral daemon and operator CLI with the reviewed lockfile:

```
cargo build --locked --release -p getaip-server -p getaip-cli
```

Cargo writes the binaries to `target/release/getaip-server` and `target/release/getaip` unless `CARGO_TARGET_DIR` is set. Verify both command surfaces without starting a service:

```
target/release/getaip-server --help
target/release/getaip --help
```

To place these commands in Cargo's binary directory instead, install the same source paths with the lockfile enforced:

```
cargo install --locked --path crates/getaip-server
cargo install --locked --path crates/getaip-cli
```

Cargo normally installs them under `$HOME/.cargo/bin`. This path installation is permitted even though the packages are not publishable.

Build connector-fleet components

A fleet adds the standalone lifecycle control plane and one host binary for each selected product boundary. The maintained product-host packages are:

Connector	Host package and binary
Cal.diy	`aip-host-cal-diy`
Hermes Agent	`aip-host-hermes-agent`
Chatwoot	`aip-host-chatwoot`
Dify	`aip-host-dify`
CrewAI	`aip-host-crewai`
Twenty	`aip-host-twenty`

For example, build a core daemon, control plane, CLI, and Cal.diy host:

```
cargo build --locked --release \
  -p getaip-server \
  -p getaip-cli \
  -p aip-connector-control-plane \
  -p aip-host-cal-diy
```

Replace or add host packages only for the connectors admitted by the target deployment. Compiling a host does not admit it, create registry credentials, or qualify its provider behavior.

For an existing bundled deployment that is being migrated, build the legacy binary explicitly and keep it separate from the new core artifact:

```
cargo build --locked --release -p getaip-server-legacy-bundled
```

Use AIP from Rust

The `aip` facade defaults to the semantic core. New applications that need the complete product-neutral implementation surface should select `full-core` at the same pinned revision:

```
[dependencies]
aip = { git = "https://github.com/getaip/core", rev =
  "e6222fc36ebd7d8a723d773606c073d5da9cd281", features = ["full-core"] }
```

The legacy `full` feature also selects product connector features. Do not use it as a shortcut for a new product-neutral integration. Product connectors remain separate packages and processes at the fleet boundary.

Add runtime dependencies only when required

The binaries can be built without starting external services. Runtime requirements depend on the selected topology:

Dependency	Required when
PostgreSQL	A deployment uses shared durable runtime state, connector registry state, or production connector-host bootstrap
NATS	The deployment enables the native NATS transport
Product credentials	A selected connector host calls its external product
Docker with Compose and BuildKit	An operator runs the repository's isolated qualification stacks

The local native quickstart uses file-backed state and does not require PostgreSQL, NATS, Docker, or product credentials.

Verify the installation

Confirm the source identity and the installed command surfaces:

```
test "$(git rev-parse HEAD)" = "$AIP_SOURCE_REVISION"
test -z "$(git status --porcelain)"
target/release/getaip-server --help >/dev/null
target/release/getaip --help >/dev/null
cargo metadata --locked --offline --no-deps --format-version 1 >/dev/null
```

If you built fleet components, run `--help` on the control-plane binary and each selected host as well. Record the source revision, Cargo metadata, build profile, and artifact digest before qualification or deployment. A tag, source checkout, compiled binary, container image, and running deployment are distinct artifact identities.

Resolve build failures

Symptom	Decision
`rustc` is older than `1.88`	Install or select a newer stable Rust toolchain, then rerun the same locked build
Cargo reports that `Cargo.lock` must change	Confirm the source revision and restore its lockfile; do not remove `--locked`
Cargo cannot work offline during final verification	Complete the locked build so dependencies are present, then rerun the verification command
The linker or filesystem reports insufficient space	Inspect the configured target directory and remove only artifacts whose rebuild cost is acceptable
A binary reports missing runtime configuration	Treat the build as installed and follow the configuration guide; rebuilding does not create credentials or deployment state
The build is being attempted on Windows	Move release-affecting work to a validated Linux environment until a Windows job and qualification scope exist

`cargo clean` acts on the target directory selected by the current invocation. Review its scope first when `CARGO_TARGET_DIR` is shared by multiple checkouts. Docker cleanup affects shared machine state and is not an installation repair.

Remove a local installation

If the binaries were installed into Cargo's binary directory, remove only the named packages:

```
cargo uninstall getaip-server
cargo uninstall getaip-cli
```

For binaries that were only built in the checkout, remove their Cargo build artifacts with:

```
cargo clean -p getaip-server -p getaip-cli
```

Remove selected control-plane or host packages by name only when they were part of the installation. Uninstalling binaries does not remove runtime databases, credential files, registry state, or product-side effects; review those assets through the applicable deployment or connector runbook.

Related pages

- Quickstart
- Connector fleet quickstart
- Profiles, transports, and connectors
- Production deployment
- Release artifacts