Build a Trusted Local Module

Use this guide to compile one reviewed, first-party capability into the same process as aipd. You will pair a context-aware action handler with an exact manifest contribution. You will then register its factory before startup and verify adm

SECTION	Build with AIP
TYPE	Guide
PROTOCOL	AIP 1.0
SOURCE	docs/guides/trusted-local-modules.md

A trusted local module is a deployment composition boundary. It is not a runtime plugin system. Adding, removing, or changing a module requires a new application build and deployment. The module shares the daemon's process, runtime stores, resource limits, and failure domain.

The examples on this page target source revision `97be86e9efedf07ecf1783b03800f683f107fb04`. This authoring pass checked the source and snippet syntax without building the workspace or starting a service.

Choose the local boundary deliberately

Use a local module only when all of these statements are true:

- the implementation is reviewed and operated as first-party daemon code;
- the module can use the daemon's release, startup, and rollback lifecycle;
- sharing one process and runtime-store handles is an accepted trust boundary;
- the complete contribution can be prepared within a bounded startup timeout;
- one invalid manifest, handler set, or route claim should fail startup.

Use the remote connector fleet when an implementation needs independent rollout, scaling, isolation, provider-specific ownership, or long-tail catalog growth. Local placement does not make an implementation safer, qualified, or production-ready. It only changes process composition and dispatch placement.

Prerequisites

You need:

- Rust `1.88` or newer and edition 2024 support;
- access to the reviewed source revision or an approved immutable mirror;
- an application that owns `AipDaemonDeployment` construction;
- a reviewed capability contract and JSON Schemas for its input and output;
- a decision that the module is required or optional during preparation;
- deployment-owned identity, approval, storage, and transport configuration.

This guide uses a pure label normalizer. It contacts no provider, stores no secret, and performs no external mutation. Production trust and storage setup remain separate deployment concerns.

1. Pin the direct dependencies

Create a binary project and pin every AIP crate to the same source revision:

```
TOML
[package]
name = "aipd-local-module"
version = "0.1.0"
edition = "2024"
rust-version = "1.88"

[dependencies]
aip-core = { git = "https://github.com/getaip/core", rev = "97be86e9efedf07ecf1783b03800f683f107fb04"
}
aip-discovery = { git = "https://github.com/getaip/core", rev =
"97be86e9efedf07ecf1783b03800f683f107fb04" }
aip-runtime = { git = "https://github.com/getaip/core", rev =
"97be86e9efedf07ecf1783b03800f683f107fb04" }
aipd = { git = "https://github.com/getaip/core", rev = "97be86e9efedf07ecf1783b03800f683f107fb04" }
async-trait = "0.1"
serde = { version = "1", features = ["derive"] }
serde_json = "1"
tokio = { version = "1", features = ["macros", "rt-multi-thread"] }
```

Every AIP workspace package has `publish = false` at the reviewed revision. Do not replace these entries with an assumed registry version. An approved vendored checkout may use path dependencies, but its revision must be recorded outside `Cargo.lock` through the deployment or SBOM process.

2. Implement the handler and module factory

Create `src/main.rs` with one complete module. The handler rejects the context-free compatibility method and accepts only the runtime-built `ActionExecutionContext` path:

```
RUST
use aip_core::{
    Action, ActionResult, ActionResultStatus, Binding, Capability, CapabilityContract,
    CapabilityId, CapabilityKind, CompensationContract, CompensationMode, DataContract,
    DataSensitivity, ExecutionContract, ExpectedCompletionMode, IdempotencyCollisionBehavior,
    IdempotencyContract, IdempotencyKeyScope, IdempotencyRequirement, Manifest, MessagePart,
    Principal, PrincipalKind, ProfileId, RetrySafety, RiskLevel, ServiceLevelContract,
    SideEffect, Stability,
};
use aip_discovery::CapabilityImplementationSupport;
use aip_runtime::{
    ActionExecutionContext, ActionHandler, RuntimeError, RuntimeResult,
};
use aipd::{
    AipDaemon, AipDaemonConfig, AipDaemonDeployment, DaemonModuleError,
    DaemonModuleFactory, DaemonServices, LocalModuleDescriptor, PreparedDaemonModule,
};
use async_trait::async_trait;
use serde::Deserialize;
use serde_json::json;
use std::{collections::BTreeSet, error::Error, sync::Arc};

const MODULE_ID: &str = "label-normalizer";
const CAPABILITY_ID: &str = "cap:example:label.normalize";
const NATIVE_HTTP_PROFILE: &str = "aip.native.http.v1";

#[derive(Clone)]
struct LabelNormalizer;

#[derive(Deserialize)]
#[serde(deny_unknown_fields)]
struct NormalizeInput {
    label: String,
}

#[async_trait]
impl ActionHandler for LabelNormalizer {
    fn implementation_support(&self) -> CapabilityImplementationSupport {
        CapabilityImplementationSupport {
            invocation: true,
```

```
            retry: true,
            ..CapabilityImplementationSupport::default()
        }
    }

    async fn handle(&self, _action: Action) -> RuntimeResult<ActionResult> {
        Err(RuntimeError::Authorization(
            "label normalizer requires trusted execution context".to_owned(),
        ))
    }

    async fn handle_with_context(
        &self,
        action: Action,
        context: ActionExecutionContext,
    ) -> RuntimeResult<ActionResult> {
        context
            .actor
            .validate(&BTreeSet::new())
            .map_err(|error| RuntimeError::Authorization(error.to_string()))?;

        let input: NormalizeInput = serde_json::from_value(action.input.clone())
            .map_err(|error| RuntimeError::Handler(error.to_string()))?;
        let normalized = input.label.split_whitespace().collect::<Vec<_>>().join(" ");
        if normalized.is_empty() {
            return Err(RuntimeError::Handler(
                "label must contain a non-whitespace character".to_owned(),
            ));
        }

        Ok(ActionResult {
            action_id: action.id,
            status: ActionResultStatus::Completed,
            output: Some(json!({ "normalized": normalized })),
            message: vec![MessagePart::text("label normalized")],
            memory_update: None,
            usage: None,
            receipt: None,
            error: None,
        })
    }
}

fn normalize_capability() -> Capability {
    Capability {
        id: CapabilityId::trusted(CAPABILITY_ID),
        name: "Normalize a label".to_owned(),
        kind: CapabilityKind::Tool,
        input_schema: json!({
            "type": "object",
            "additionalProperties": false,
            "required": ["label"],
            "properties": {
                "label": { "type": "string", "minLength": 1, "maxLength": 256 }
            }
        }),
        output_schema: Some(json!({
            "type": "object",
            "additionalProperties": false,
            "required": ["normalized"],
            "properties": {
                "normalized": { "type": "string", "minLength": 1, "maxLength": 256 }
            }
        })),
        description: Some("Collapses whitespace in one label.".to_owned()),
        risk: Some(RiskLevel::Low),
        stability: Some(Stability::Stable),
        cost: None,
        auth: None,
        bindings: vec![Binding {
            profile: ProfileId::from(NATIVE_HTTP_PROFILE),
            metadata: [
                ("method".to_owned(), json!("POST")),
                ("path".to_owned(), json!("/aip/v1/messages")),
                (
                    "message_type".to_owned(),
                    json!("aip.core.v1.action"),
```

```
                ),
            ]
            .into_iter()
            .collect(),
        }],
        requires_human_approval: Some(false),
        contract: Some(CapabilityContract {
            side_effects: vec![SideEffect::Read],
            idempotency: IdempotencyContract {
                requirement: IdempotencyRequirement::Optional,
                collision_behavior: IdempotencyCollisionBehavior::ReturnOriginalResult,
                key_scope: IdempotencyKeyScope::Capability,
                ttl_ms: None,
            },
            execution: ExecutionContract {
                supports_sync: true,
                supports_async: false,
                supports_streaming: false,
                supports_cancel: false,
                supports_retry: true,
                expected_completion: ExpectedCompletionMode::Sync,
                retry_safety: RetrySafety::Safe,
            },
            data: DataContract {
                sensitivity: DataSensitivity::Internal,
                contains_pii: false,
                redaction_required: false,
                residency: None,
                retention: None,
            },
            credentials: None,
            approval: None,
            sla: Some(ServiceLevelContract {
                expected_latency_ms: Some(10),
                timeout_ms: Some(1_000),
                async_expected: false,
                max_queue_delay_ms: Some(100),
                availability_target: None,
            }),
            transaction: None,
            compensation: Some(CompensationContract {
                mode: CompensationMode::NotRequired,
                compensation_capability_id: None,
                compensation_window_ms: None,
                requires_approval: false,
            }),
        }),
    }
}

#[derive(Clone)]
struct LabelNormalizerModule;

#[async_trait]
impl DaemonModuleFactory for LabelNormalizerModule {
    fn descriptor(&self) -> LocalModuleDescriptor {
        LocalModuleDescriptor::required_static(MODULE_ID)
    }

    async fn prepare(
        &self,
        services: &DaemonServices,
    ) -> Result<PreparedDaemonModule, DaemonModuleError> {
        let capability = normalize_capability();
        let manifest = Manifest {
            manifest_version: "aip-manifest/v1".to_owned(),
            agent: Principal::new(services.service_id().clone(), PrincipalKind::Agent),
            capabilities: vec![capability.clone()],
            profiles: vec![ProfileId::from(NATIVE_HTTP_PROFILE)],
            resources: Vec::new(),
            channels: Vec::new(),
            security: None,
            governance: None,
            limits: None,
            compatibility: None,
            extensions: None,
        };
```

```
        PreparedDaemonModule::new(self.descriptor(), manifest)
            .with_handler(capability, Arc::new(LabelNormalizer))
    }
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn Error>> {
    let config = AipDaemonConfig::default();
    let bind = config.bind;
    let deployment = AipDaemonDeployment::default()
        .with_module_factory(LabelNormalizerModule);
    let daemon = AipDaemon::new_with_deployment(config, deployment).await?;

    let admitted = daemon
        .manifest()
        .capabilities
        .iter()
        .any(|capability| capability.id.as_str() == CAPABILITY_ID);
    if !admitted {
        return Err(std::io::Error::other("local capability was not admitted").into());
    }

    daemon.serve(bind).await?;
    Ok(())
}
```

The descriptor is static and performs no I/O. `prepare` receives only the daemon service principal and cloned runtime-store handles. It does not receive the gateway router, active-action map, callback dispatcher, or policy engine.

The handler's implementation support must match its capability contract. Invocation and safe retry are enabled here. Cancellation, streaming, transactions, reconciliation, compensation, approval, and credential resolution remain disabled.

3. Keep the manifest and handler set exact

`PreparedDaemonModule::with_handler` records a handler by capability ID. The daemon later compares the complete executable capability set with the complete handler set. Every non-resource capability needs exactly one handler, and a resource capability cannot receive an action handler.

The module manifest contributes capabilities, profiles, resources, and channels to the daemon manifest. The daemon retains its own security, governance, limits, and principal configuration. It also records a bounded module summary under the combined manifest's connector extensions.

Do not treat a module manifest as a way to override daemon security. Configure identity resolvers, approval authority, storage, callback policy, and transports in the deployment that owns `AipDaemonConfig` and `AipDaemonDeployment`.

4. Register the factory before daemon construction

`with_module_factory` adds the concrete factory to the frozen deployment composition. `AipDaemon::new_with_deployment` prepares and admits every factory before it returns a daemon. `serve` binds the listener only after that constructor has succeeded.

The example uses default daemon security solely to keep module composition visible. Default native AIP requests still require signed envelopes and trusted signer configuration. Add production trust and durable storage through the owning deployment; do not weaken authentication to test the module.

Generate the lockfile, compile the application, and start it in the intended test environment:

```
SH
cargo generate-lockfile
cargo check --locked
cargo run --locked
```

Treat these commands as instructions for the consumer project. They were not executed during this source-only documentation pass.

5. Verify admission and readiness

In another terminal, query the native manifest and readiness endpoint:

```
SH
curl -fsS http://127.0.0.1:8080/aip/v1/manifest \
  | jq -e '.capabilities[] | select(.id == "cap:example:label.normalize")'

curl -fsS http://127.0.0.1:8080/ready \
  | jq -e '.modules["label-normalizer"] == {
      "required": true,
      "ready": true,
      "detail": "module prepared and admitted"
    }'
```

Both commands must exit with status `0`. A `ready: true` module status means that preparation and admission succeeded. It does not prove provider health, external side effects, durability, conformance, qualification, or production readiness.

If the module registers a readiness connector, gateway readiness evaluates that connector separately. Module admission status is not a substitute for its readiness result.

6. Understand the startup decision

The daemon applies this sequence before publishing the combined manifest:

1. reject more than 64 factories;
2. validate every static ID and timeout;
3. sort factories by module ID and reject duplicate IDs;
4. run each `prepare` future under its declared timeout;
5. require the returned descriptor to equal the static descriptor;
6. validate manifest size and exact handler coverage;
7. reject capability, connector, and HTTP-route ownership conflicts;
8. merge accepted contributions and construct the gateway atomically;
9. register readiness connectors and delegation routers;
10. bind the HTTP listener only when application code calls `serve`.

Required and optional descriptors differ only at step 4. An error or timeout from `prepare` stops startup for a required module. The same result skips an optional module and records `required: false, ready: false` in `/ready`.

Optional does not suppress an invalid ID, invalid timeout, changed descriptor, invalid manifest, handler mismatch, or ownership conflict. Those errors still stop daemon construction because the composition cannot be admitted safely.

7. Stay within the module limits

The reviewed implementation applies these hard bounds:

Boundary	Limit	Design consequence
Modules per daemon	64	Prefer the remote fleet for a growing connector catalog

Boundary	Limit	Design consequence
Module ID	1–64 characters	Use lower-case ASCII letters, digits, `.`, `_`, or `-`
Preparation timeout	1–300,000 ms	Default is 30,000 ms; keep startup work bounded
Serialized manifest	4 MiB per module	Keep product catalogs and large generated data outside the contribution
Executable handlers	4,096 per module	Every callable capability still needs one exact handler
HTTP route claims	256 per mount	Declare each method and path for conflict detection
HTTP path	512 characters	Keep every module route below `/connectors/`
Error detail created with `DaemonModuleError::new`	1,024 characters	Never place credentials or provider payloads in the message

`DaemonModuleError::new` truncates its message, but truncation is not redaction. Return a stable error code and a bounded operational description without secret material.

8. Add optional contributions only when owned

A prepared module can contribute more than action handlers:

Contribution	API	Ownership requirement
Runtime-store access during preparation	`DaemonServices::runtime_stores`	Use only stores owned by the daemon lifecycle
Readiness connector	`with_connector_arc`	Use a stable, unique connector ID and bounded readiness checks
Native delegation router	`with_delegation_router_arc`	Return ownership only for the remote delegations the module can route
State-complete HTTP router	`with_http_mount`	Declare every method and path, all under `/connectors/`

An HTTP mount carries both an Axum router and explicit route claims. The daemon uses the claims for deterministic conflict checks. Keep the claims identical to the router's actual routes; the constructor does not infer them from Axum.

Do not use a local HTTP mount to claim a core route such as `/aip/v1/messages`. Module-owned routes are restricted to `/connectors/`.

9. Review the trust boundary

Before accepting a module, confirm all of the following:

- maintainers review it as part of the daemon artifact;
- `descriptor` is deterministic and never contacts a provider;
- `prepare` is bounded, cancellation-safe, and contains no unbounded discovery;
- every handler uses `handle_with_context` for trusted runtime state;
- a trust-dependent handler rejects the context-free `handle` path;
- identity comes from `context.actor`, never from action input;
- tenant, credential, approval, transaction, and redaction data come only from

their matching context fields;

- logs and errors omit action payloads, credentials, and provider responses;
- manifest contracts describe only behavior the handler implements;

- module and remote fleet implementations never claim the same capability ID;
- the deployment accepts same-process memory, CPU, store, and crash exposure.

`ActionExecutionContext` is deliberately non-serializable. It contains the transport-authenticated actor, optional verified tenant, opaque credential handle, deadline, cancellation token, idempotency reservation, and approval. It also carries transaction state, checkpoint publishers, a stream, trusted trace identifiers, and the redaction policy. Do not reconstruct it from an AIP payload.

Resolve common failures

Symptom or code	Meaning	Bounded action
`aipd.module.invalid_id`	The module ID violates the static format	Fix the compile-time ID; do not normalize it silently
`aipd.module.invalid_timeout`	The timeout is zero or exceeds five minutes	Choose a bounded value within the supported range
`aipd.module.duplicate_id`	Two factories own the same module ID	Remove one owner or assign a stable distinct ID
`aipd.module.startup_timeout`	Preparation exceeded its descriptor timeout	Bound the preparation work; inspect whether the module must remain required
`aipd.module.descriptor_changed`	`prepare` returned a different descriptor	Return the exact static descriptor from the prepared module
`aipd.module.manifest_limit`	The serialized contribution exceeds 4 MiB	Move generated or tenant-specific catalog data out of the local manifest
`aipd.module.handler_manifest_mismatch`	Callable capabilities and handlers differ	Add the exact missing handler or remove the undeclared handler
`aipd.module.capability_conflict`	Core or another module owns the capability ID	Preserve one canonical owner and change the manifest before retrying startup
`aipd.module.connector_conflict`	Two modules return the same readiness connector ID	Preserve one connector owner or assign a stable distinct ID
`aipd.module.reserved_http_route`	A module route is outside `/connectors/`	Move it under the module namespace; never shadow a core route
`aipd.module.http_route_conflict`	Two modules claim the same method and path	Keep one owner and update both claims and router code together
Daemon starts without an optional capability	Its `prepare` call failed or timed out	Inspect `/ready`, fix the bounded failure, and redeploy; do not invoke an undiscovered capability

Preserve the first startup error. Repeated restarts cannot repair a deterministic manifest or ownership conflict and can hide the original evidence.

Roll back the compiled composition

A local module cannot be unloaded or hot-swapped. To roll it back:

1. stop routing new work to the affected daemon artifact;
2. preserve the startup error and deployed source identity;
3. remove the matching `.with_module_factory(...)` registration;
4. rebuild from the last approved dependency and configuration set;
5. deploy the replacement artifact through the normal rollout boundary;
6. verify that the capability is absent from the manifest;
7. verify that the old module ID is absent from `/ready`;
8. reconcile any actions accepted before the old process stopped.

Removing the factory changes discovery after redeployment. It does not cancel, reverse, or settle work already persisted by the previous process.

Related documentation

- Use the Rust SDK to select and pin the required crate boundaries.
- Read Capabilities and contracts before defining the manifest and handler support matrix.
- Read Actions and sessions for durable lifecycle, idempotency, cancellation, and unknown outcomes.
- Read Identity and trust before consuming authenticated actor, tenant, credential, or approval context.
- Read Profiles, transports, and connectors before choosing local placement instead of the remote fleet.