

MISP

INTELLIGENCE SHARING

Contributing MISP modules with an AI agent

From a precise brief to a tested pull request

CIRCL / MISP Project

https://www.misp-project.org/

Reference snapshot: 29 October 2026

What a successful contribution contains

- A module that fits existing discovery and execution.
- Correct MISP inputs, outputs, configuration and provenance.
- Tests that demonstrate behavior, not just Python syntax.
- Useful metadata, documentation and a focused pull request.

The agent assists development

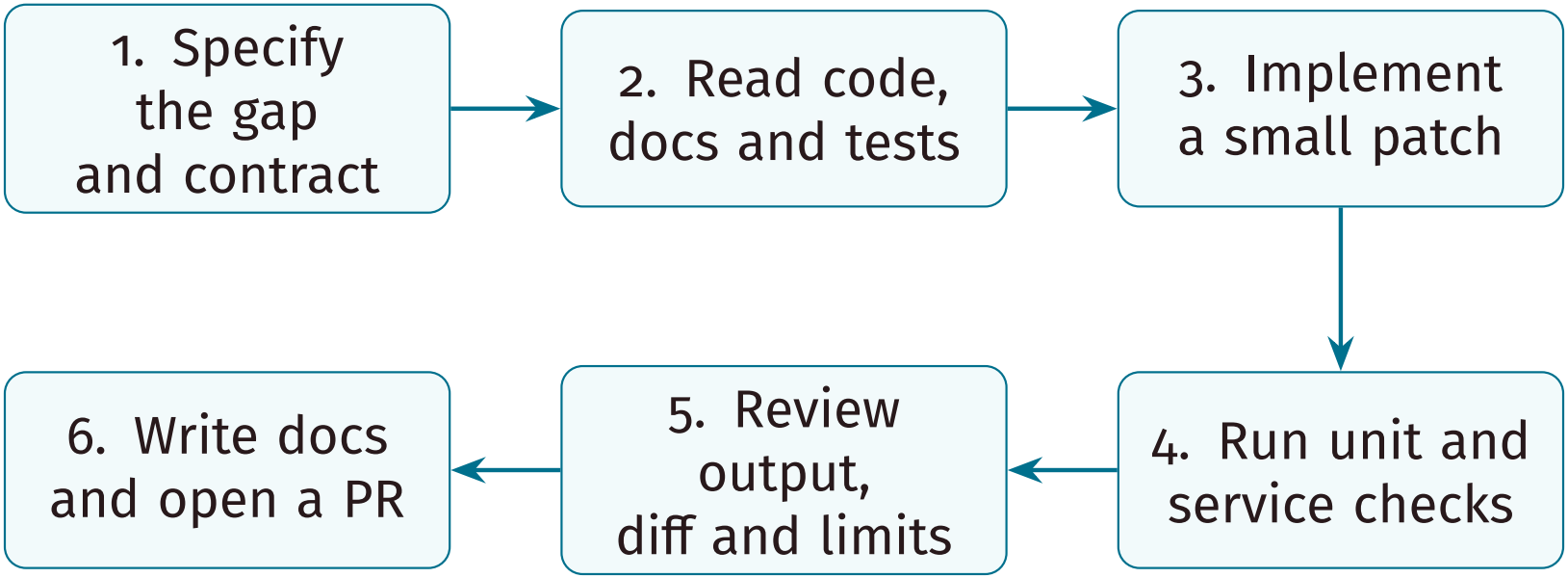
It is not a new module family or a reason to add a model service to the runtime implementation.

Keep ownership and evidence explicit

Contributor	Coding agent	Maintainer
Defines the need and boundaries.	Inspects code and prepares a plan and patch.	Assesses fit and maintenance cost.
Checks contracts and reviews results.	Runs permitted checks and reports actual outcomes.	Reviews the patch and evidence.
Owns the submitted contribution.	Identifies assumptions and limits.	Decides whether to merge.

A confident explanation from an agent is not execution evidence.

A repeatable development loop



Reference snapshot: misp-modules 3a903305cf37, reviewed 29 October 2026.

Read the existing contract

Give the agent the right repository context

- Contributor instructions, dependency declarations and CI.
- Loader and dispatcher: `misp_modules/__init__.py` and `__main__.py`.
- A recent module of the **same family**, plus its tests.
- Existing shared libraries and target-format builders.
- Official API/schema documentation, version and examples.

Is a new module needed?

Search existing capabilities before implementing a duplicate integration.

Select a family before writing code

Directory	Metadata type	Typical contract
`expansion`	expansion / hover	Observable or attribute to enrichment results.
`import_mod`	import	File, paste or report to parsed MISP data.
`export_mod`	export	Supplied MISP data to an encoded artifact.
`action_mod`	action	Workflow data and parameters to an operation/result.

Match the specific module's schema; this is not one universal request format.

Preserve the module interface

- `introspection()`: supported input/output or family-specific setup.
- `version()`: accurate capabilities, configuration and metadata.
- `dict_handler(request=...)`: dictionary execution path.
- `handler(q=...)`: supported legacy JSON execution path.
- Return fresh result/error objects; keep request state local.

Read the fallback behavior

The reviewed dispatcher catches `AttributeError` for legacy fallback. Do not use that exception as input validation.

Write down the request and result schema

- Supported attributes, event envelopes or file inputs?
- Settings in `config`, `params` or import options?
- Simplified expansion or `format: misp_standard`?
- Attributes, objects, an encoded file or action status?
- Missing data, empty results and partial failures?

Use existing MISP semantics

Map the provider's response into valid types, object templates and the selected family's contract.

Keep provenance and detection intent

- Preserve the source UUID and original evidence.
- Use PyMISP to build and serialize structured objects.
- Link derived objects to the correct source.
- Choose valid types and object relations.
- Do not silently promote contextual data to IDS indicators.

Reference exercise

Keep the original attribute; add a normalized `domain-ip` object with a `derived-from` reference.

Dependencies and configuration are not incidental

- Use the service's actual Python environment and dependency workflow.
- Add necessary dependencies without unrelated lockfile upgrades.
- Metadata describing a dependency does not install it.
- Distinguish request config, service defaults and module-specific environment variables.
- Keep credentials and private incidents out of prompts and fixtures.

Direct the agent and test behavior

A useful brief needs an explicit contract

1. Purpose, gap and unique identifier.
2. Family, supported inputs and expected outputs.
3. Official provider/schema version and configuration keys.
4. Side effects and permitted execution boundaries.
5. Acceptance tests, error cases and required evidence.

Reusable task prompt

Start with `AGENT_BRIEF.md`. It is a prompt to adapt, not an automatically installed agent policy.

Worked exercise: normalize a DNS name locally

- Input: `domain/hostname`, value and source UUID.
- Normalize case, one root dot and standard-library IDNA.
- Reject malformed labels, unsupported types, IP literals and bad UUIDs.
- Preserve the input; add a referenced MISP-standard object.
- No DNS lookup, provider account, model call or runtime agent framework.

Training scope

This is a contract exercise, not a proposal to add another normalizer upstream. A contribution must address a real project gap.

An acceptance request the agent can test

```
{
  "module": "training_domain_normalize",
  "attribute": {
    "type": "domain",
    "value": "B\u00dcCHER.Example.",
    "uuid": "13d8d18b-784a-42a0-9a58-a9cddbfda561",
    "to_ids": true,
    "comment": "Synthetic training source"
  }
}
```

Expected: `xn--bcher-kva.example` in a `domain-ip` object linked to the original source UUID.

Ask for small, reviewable implementation steps

1. Metadata and introspection.
2. Pure parsing/normalization and validation.
3. MISP result construction with references.
4. A thin provider boundary, if the task needs one.
5. Tests and documentation for the actual contract.

Inspect the diff

Reject unrelated framework changes, dependency churn, import-time networking and invented endpoint behavior.

If a module calls a provider, test the boundary

- Mock the provider client or HTTP boundary where it is used.
- Use documented fixtures: valid, empty and malformed responses.
- Test authentication errors, rate limits and bounded timeouts.
- Preserve references across multiple and partial results.
- Verify scope and acknowledgements for operations that write.

Keep output construction real

Mock external I/O, not the PyMISP objects that assertions need to inspect.

Make tests challenge the implementation

Dimension	What can go wrong
Input/schema	Missing fields, unsupported types, Unicode and length boundaries.
MISP semantics	Wrong relations, lost source UUIDs, changed detection flags.
State isolation	Caller or request mutates another result or shared metadata.
Errors	Exceptions hidden as empty success; errors ignored at HTTP 200.
Deployment	Service imports another package or skips the new file.

Report test levels honestly

- **Unit**: validation, parsing, serialization and isolation.
- **Service**: discovery and real `/query` requests.
- **MISP**: family settings, user access and UI ingestion.
- **Provider/target**: authorization, acknowledgements and activation.

Keep evidence scoped

Record commands, exit codes and counts. Distinguish passed, failed, skipped and unrun checks.

Verify and contribute

Check what the service actually imports

- Use an editable package install or explicit custom root.
- Install dependencies in the service environment.
- Restart the correct process.

```
# From this companion deck's directory:
misp-modules -l 127.0.0.1 -p 6666 \
  --custom "$PWD/examples/custom"
```

The root contains `expansion/training_domain_normalize.py`.

Verify discovery and dispatch

```
curl --fail --silent --show-error \
  http://127.0.0.1:6666/modules |
  jq '.[] | select(.name == "training_domain_normalize")'

python3 examples/verify_module.py \
  --url http://127.0.0.1:6666 -v
```

- Check family, capabilities, input format and config schema.
- Verify valid results and invalid-input errors through HTTP.
- Inspect `/openapi.json`, not just the handler.

HTTP 200 is not the acceptance criterion

- Inspect `error`/`errors` and the actual output envelope.
- Check references and source fields after serialization.
- Decode exports and inspect the target format.
- Check action outcomes appropriate to the operation.
- Verify MISP enablement separately from service discovery.

Use the project's quality and documentation workflow

- Read current CI, Poetry groups, extras and native prerequisites.
- Run relevant tests and formatting/lint under project configuration.
- Broaden checks when changes or failures warrant it.
- Supply complete metadata and user documentation.
- Keep dependency/lock edits tied to the implementation.

The brief supplies command templates; substitute actual paths and family.

What the pull request should explain

1. Integration gap and resulting behavior.
2. Input/output examples and configuration, without secrets.
3. Provider/schema version and side effects.
4. Actual validation, commands, counts and limitations.
5. Dependencies and documentation needed by users.

Keep it focused

Include the implementation, tests and documentation a maintainer needs. Remove incidental edits and generated noise.

Human review is part of the workflow

- Compare mappings with official API/schema documentation.
- Read validation, errors, references and config behavior.
- Inspect tests for missing cases, not only their exit status.
- Check for unexpected target writes or runtime model calls.
- Reconsider a duplicate or unnecessary upstream integration.

Workshop: brief, implement, verify

1. Adapt `AGENT_BRIEF.md` to the exercise.
2. Ask for a contract and acceptance plan.
3. Implement in a separate local directory; use the reference as an answer key.
4. Run unit and service checks and review source/object relationships.
5. Draft a PR description stating what was and was not tested.

The logic to preserve

1. Existing loader, family and dispatch contracts.
2. Types, object templates, source references and detection intent.
3. Explicit configuration, bounded I/O and useful failures.
4. Repeatable tests and installation/discovery evidence.
5. A contribution that a maintainer can review.

Resources for the next contribution

- `README.md`: setup, exercise specification and source audit.
- `AGENT_BRIEF.md`: reusable task prompt.
- `examples/`: reference, request and verifier.
- MISP/misp-modules and its current contribution guide.
- Module and installation documentation.
- The companion *MISP modules* introduction and local API lab.