ChatGPT

Clawffee Repository Structure & Interfaces

This document describes how the `Clawffee` codebase is organised when operating as an OpenClaw plugin. OpenClaw expects plugins to follow a clear, modular structure: each plugin must declare its metadata, expose operators in a discoverable way, include tests, and provide sufficient documentation. The layout presented here intentionally mirrors the conventions used throughout the OpenClaw project so that the repository will feel familiar to contributors who have worked on other OpenClaw-compatible operators.

Top-Level Layout

The following tree shows the high-level arrangement of folders and files within the repository. Although the names and locations mirror typical Python project conventions, they intentionally reflect OpenClaw's expectations about where to find plugin declarations, operator implementations, tests, and documentation.

```
Clawffee/
├── openclaw_plugin.yaml
│   Metadata that declares the plugin name, version, description, and lists the
available operators.  OpenClaw reads this file to discover and load the plugin
at runtime.
├── README.md
│   An approachable overview of the project: what it does, how to install it,
and how to run the sample operators.  Contributors and users should be able to
get started without digging into the internals.
├── LICENSE
│   The chosen license for the project.
├── plugins/
│   Container for all OpenClaw operators exposed by this plugin.  Each operator
lives in its own module under the `clawffee` namespace.
│   └── clawffee/
│       ├── __init__.py  – registers operators for discovery by calling the
appropriate `register` function from OpenClaw's API.
│       ├── baseline.py  – implementation of the baseline summariser operator.
│       └── tot.py       – implementation of the tree-of-thought summariser
operator.
├── tests/
│   Unit and integration tests.  Test files mirror the operator modules and use
the pytest framework.  Running `pytest` at the repository root should execute
these tests to ensure that the plugin still satisfies OpenClaw's contracts.
├── docs/
│   Additional documentation, requirements, and design artefacts for
contributors.  These Markdown files can be rendered by GitHub or any other
Markdown viewer.
```

```
│   ├── Clawffee_MVP_Spec.md
│   ├── Clawffee_Requirements.md
│   └── Clawffee_PDR.md
└── scripts/
    Helper scripts for local development.  For example, `evaluate.py` runs
sample evaluations against the baseline and tree-of-thought operators.
```

Key Components

`openclaw_plugin.yaml`

This YAML file is the entry point for OpenClaw's plugin loader. It provides high-level metadata about the plugin and enumerates the operators contained within it. At a minimum the file must define:

- `id` – a unique identifier for the plugin (`clawffee`).
- `description` – a brief description of the plugin's purpose.
- `authors` – a list of author names and optional contact information.
- `version` – semantic version string.
- `operators` – a list of dictionaries specifying the module and class for each operator.
- `capabilities` – a dictionary listing the pipeline stages supported by the plugin and the type of operation (analysis, synthesis, etc.).
- `supported_request_types` – a list of MIME types accepted by the operators. For Clawffee this is `text/plain`.

Example:

```
id: clawffee
description: Summarisation operators for meeting transcripts with baseline and
Tree-of-Thought modes.
authors:
  - name: Stephan (hodlneer)
version: 0.1.0
operators:
  - module: plugins.clawffee.baseline
    class: BaselineSummaryOperator
  - module: plugins.clawffee.tot
    class: TreeOfThoughtSummaryOperator
capabilities:
  stage:
    - analysis
  operation: summarisation
supported_request_types:
  - text/plain
```

`plugins/clawffee/__init__.py`

This file is responsible for registering the operators with OpenClaw. It imports both operator classes and exposes them in a list called `__all__`. A minimal example:

```
from .baseline import BaselineSummaryOperator
from .tot import TreeOfThoughtSummaryOperator

__all__ = [
    BaselineSummaryOperator,
    TreeOfThoughtSummaryOperator,
]
```

`plugins/clawffee/baseline.py`

This module implements the `BaselineSummaryOperator` class. It subclasses `OpenClawOperatorBase` and overrides the required methods. Key methods include:

- `name(self) -> str` – returns `'baseline-summary'`.
- `version(self) -> str` – returns `'0.1.0'`.
- `profile(self) -> Dict[str, Any]` – returns a human-readable description of what the operator does and may include usage instructions.
- `capabilities(self) -> Dict[str, Any]` – returns `{"stage": ["analysis"], "operation": "summarisation"}`.
- `default_config(self) -> Dict[str, Any]` – returns default model parameters.
- `transform(self, payload: Dict[str, Any], config: Dict[str, Any]) -> Dict[str, Any]` – core logic. Reads `payload["text"]`, constructs a prompt, sends it to the LLM via OpenClaw's client and returns `{"summary": result}`.

The class should avoid side effects and return pure data structures.

`plugins/clawffee/tot.py`

This module implements the `TreeOfThoughtSummaryOperator`. It follows the same structure as the baseline operator but adds additional internal helper methods to segment the input, build reflection prompts and compose the final narrative. It should return both the final `summary` and a list of `branches` with the intermediate question–answer pairs.

`tests/`

Unit tests live here. Each operator should have corresponding tests verifying that it returns the expected dictionary shape given a sample input. Tests can mock the LLM client to return deterministic results.

`docs/`

This folder contains design documents and specifications. Contributors should update these files whenever functionality changes. Key files include:

- `Clawffee_MVP_Spec.md` – high-level specification of the operator's purpose and features.
- `Clawffee_Requirements.md` – functional and non-functional requirements.
- `Clawffee_PDR.md` – project design review; architecture diagrams, rationale and risk analysis.

`scripts/`

Utility scripts used during development or evaluation. For example, a script to run the operator locally outside of OpenClaw, or a simple evaluation harness to compute ROUGE scores.

Interfaces

Operator Interface

Each operator implements the abstract methods defined on `OpenClawOperatorBase`. The most important interface is the `transform` method:

```
def transform(self, payload: Dict[str, Any], config: Dict[str, Any]) ->
Dict[str, Any]:
    """Transform the input payload into a summarisation result.

    Args:
        payload: Input data with at least a `text` field containing the full
transcript.
        config: User-supplied configuration overriding the defaults.

    Returns:
        A dictionary containing at minimum a `summary` key.  The ToT operator
also returns `branches`.
    """
    raise NotImplementedError
```

OpenClaw will pass in a `payload` dictionary and expects a dictionary in return. Operators should not assume the presence of any other keys beyond `text` unless declared in the plugin documentation.

Configuration Interface

The `default_config` method returns default parameters. When invoking an operator, users may supply a partial configuration dictionary; any unspecified values fall back to defaults. Operators should merge the provided config with the defaults at runtime:

```
cfg = {**self.default_config(), **config}
```

Common configuration keys include:

- `model` – LLM model name (e.g. `gpt-4-1106-preview`).
- `max_tokens` – maximum tokens allowed for the model response.
- `section_size` – number of characters per Tree-of-Thought section.
- `reflection_questions` – list of questions used for the ToT operator.

Testing Interface

Tests should import the operator classes and call `transform` directly with dummy payloads and configurations. The LLM client can be mocked or stubbed to return a predetermined response. The tests should verify that the output dictionary has the expected keys (`summary`, `branches` for ToT) and that the narrative integrates the mocked answers correctly.

Contributing

When adding new features or operators, please update both the code and the relevant documentation in `docs/`. Maintain semantic versioning by incrementing the `version` field in `openclaw_plugin.yaml` whenever you introduce breaking changes or new functionality. New operators should be accompanied by corresponding unit tests under `tests/`.

By adhering to this structure and interface specification, Clawffee will remain maintainable and compatible with the OpenClaw framework as it evolves.
