```
"""
OpenClaw Adapter — E-T Systems Module Integration for OpenClaw Skills

Bridges the framework-agnostic NGEcosystem to OpenClaw's skill interface.
Any E-T Systems module that wants to function as an OpenClaw skill vendors
this file alongside ng_ecosystem.py and ng_lite.py.

The adapter provides the standard OpenClaw skill vocabulary:
    on_message(text)  → process one conversation turn
    recall(text)      → semantic context retrieval
    stats()           → telemetry for OpenClaw's skill system

It also writes structured JSONL events to {workspace}/memory/events.jsonl
so OpenClaw's memory system can parse activity without stdout scraping.

Usage (inside a module's openclaw hook file):

    from openclaw_adapter import OpenClawAdapter

    class TrollGuardHook(OpenClawAdapter):
        MODULE_ID = "trollguard"
        SKILL_NAME = "TrollGuard Security"
        WORKSPACE_ENV = "TROLLGUARD_WORKSPACE_DIR"
        DEFAULT_WORKSPACE = "~/.openclaw/trollguard"

        def _embed(self, text: str) -> "np.ndarray":
            # Return your module's embedding for text.
            # If you have no embedder, use the fallback:
            return self._hash_embed(text)

        def _module_on_message(self, text: str, embedding: "np.ndarray") -> dict:
            # Module-specific processing (scan, route, classify, etc.)
            # Return a dict to merge into the on_message result.
            return {}

        def _module_stats(self) -> dict:
            # Module-specific stats to merge into stats().
            return {}

    # Singleton wiring — identical across all modules:
    _INSTANCE = None

    def get_instance():
        global _INSTANCE
        if _INSTANCE is None:
            _INSTANCE = TrollGuardHook()
        return _INSTANCE

Then in SKILL.md, autoload: true and set the hook to this file.
OpenClaw calls get_instance().on_message(text) on every turn.

Canonical source: https://github.com/greatnorthernfishguy-hub/NeuroGraph
License: AGPL-3.0

# ---- Changelog ----
# [2026-02-22] Claude (Sonnet 4.6) — Initial creation.
#   What: OpenClawAdapter ABC — base class for all E-T Systems module
#         OpenClaw hooks.  Provides on_message(), recall(), stats(),
#         memory event logging, and embedding fallback.  Subclasses
#         supply MODULE_ID, WORKSPACE_ENV, _embed(), _module_on_message(),
#         and _module_stats().
#   Why:  Each module was implementing the OpenClaw hook pattern from
#         scratch.  This standardizes the interface so OpenClaw sees
#         identical vocabulary from every E-T Systems skill, and modules
#         only implement what's unique to them.
#   Settings: auto_save_interval=10 messages (matches NeuroGraph's own
#         hook; balances persistence safety vs I/O overhead).
#   How:  ABC with abstract properties + three hook methods.  NGEcosystem
#         is initialized in __init__ via ng_ecosystem.init().  Memory
#         event log written to {workspace}/memory/events.jsonl — OpenClaw
#         standard location, same as NeuroGraph's own hook.
# -------------------
"""

from __future__ import annotations

import json
import logging
import os
import time
from abc import ABC, abstractmethod
from pathlib import Path
from typing import Any, Dict, Optional

import numpy as np

logger = logging.getLogger("openclaw_adapter")


class OpenClawAdapter(ABC):
    """Base class for OpenClaw skill hooks over NGEcosystem.
    Subclass this in each module.  Override:
        MODULE_ID           — module identifier string (required)
        SKILL_NAME          — human-readable name for logs/stats (required)
        WORKSPACE_ENV       — env var name for workspace dir (required)
        DEFAULT_WORKSPACE   — fallback workspace path (required)
        _embed(text)        — return np.ndarray embedding for text (required)
        _module_on_message  — module-specific processing per message (optional)
        _module_stats       — module-specific stats dict (optional)

    Do NOT override on_message(), recall(), or stats() directly.
    """

    # --- Override in subclass ---
    MODULE_ID: str = ""
    SKILL_NAME: str = ""
    WORKSPACE_ENV: str = ""
    DEFAULT_WORKSPACE: str = ""

    AUTO_SAVE_INTERVAL: int = 10  # messages between auto-saves

    def __init__(self) -> None:
        if not self.MODULE_ID:
            raise ValueError("Subclass must set MODULE_ID")

        # Workspace
        ws_raw = os.environ.get(self.WORKSPACE_ENV, self.DEFAULT_WORKSPACE)
        self._workspace = Path(ws_raw).expanduser()
        self._workspace.mkdir(parents=True, exist_ok=True)
        (self._workspace / "memory").mkdir(exist_ok=True)
        self._events_log = self._workspace / "memory" / "events.jsonl"

        # NGEcosystem (shared with the rest of the module)
        import ng_ecosystem
        self._eco = ng_ecosystem.init(
            module_id=self.MODULE_ID,
            state_path=str(self._workspace / "ng_lite_state.json"),
        )

        self._message_count = 0
        self._start_time = time.time()
        logger.info("[%s] OpenClawAdapter ready (tier %d)", self.MODULE_ID, self.

    # ------------------------------------------------------------------
    # Abstract methods — module must implement
    # ------------------------------------------------------------------
```

```
@abstractmethod
def _embed(self, text: str) -> np.ndarray:
    """Return a normalized np.ndarray embedding for text.

    If your module has no embedder, call self._hash_embed(text) as
    a zero-dependency fallback.
    """
    ...

def _module_on_message(self, text: str, embedding: np.ndarray) -> Dict[str, A
    """Module-specific processing for each message.

    Override to run your module's core logic (scan, route, classify…).
    The return dict is merged into the on_message() result.
    Default: no-op.
    """
    return {}

def _module_stats(self) -> Dict[str, Any]:
    """Module-specific stats to merge into stats().  Default: no-op."""
    return {}

# ------------------------------------------------------------------
# OpenClaw skill interface
# ------------------------------------------------------------------

def on_message(self, text: str) -> Dict[str, Any]:
    """Process one OpenClaw conversation turn.

    Called by OpenClaw on every user message.  Embeds text, runs
    ecosystem learning, runs module-specific processing, logs event.

    Returns:
        {
            "status":       "ingested" | "skipped",
            "tier":         int,
            "tier_name":    str,
            "message_count": int,
            "module_results": dict  (from _module_on_message),
            "recommendations": list,
            "novelty":      float,
        }
    """
    if not text or not text.strip():
        return {"status": "skipped", "tier": self._eco.tier, "message_count":

    self._message_count += 1
```

```
        embedding = self._embed(text)

        # Record to ecosystem (Tier 1/2/3 transparently)
        self._eco.record_outcome(
            embedding,
            target_id=f"message:{self._message_count}",
            success=True,
            metadata={"source": "openclaw", "module": self.MODULE_ID},
        )

        # Module-specific processing
        module_results = self._module_on_message(text, embedding)

        # Context from ecosystem
        ctx = self._eco.get_context(embedding)

        result = {
            "status": "ingested",
            "tier": self._eco.tier,
            "tier_name": self._eco.tier_name,
            "message_count": self._message_count,
            "module_results": module_results,
            "recommendations": ctx["recommendations"],
            "novelty": ctx["novelty"],
        }

        self._write_event("message", result)

        # Auto-save
        if self._message_count % self.AUTO_SAVE_INTERVAL == 0:
            self._eco.save()

        return result

    def recall(self, query: str, top_k: int = 5) -> Dict[str, Any]:
        """Retrieve cross-module context for a query.

        Called by OpenClaw when context retrieval is requested.

        Returns:
            {
                "tier":            int,
                "recommendations": list of (target_id, confidence, reasoning),
                "novelty":         float,
                "ng_context":      str | None  (Tier 3 only),
            }
        """
        embedding = self._embed(query)
        ctx = self._eco.get_context(embedding, top_k=top_k)
        self._write_event("recall", {"query": query[:200], **ctx})
        return ctx

    def stats(self) -> Dict[str, Any]:
        """Return unified stats for OpenClaw skill telemetry."""
        eco = self._eco.stats()
        module = self._module_stats()
        uptime = time.time() - self._start_time
        return {
            "skill": self.SKILL_NAME,
            "module_id": self.MODULE_ID,
            "uptime_seconds": round(uptime, 1),
            "message_count": self._message_count,
            "workspace": str(self._workspace),
            "ecosystem": eco,
            "module": module,
        }

    # ------------------------------------------------------------------
    # Memory event logging (OpenClaw standard)
    # ------------------------------------------------------------------

    def _write_event(self, event_type: str, data: Dict[str, Any]) -> None:
        """Append a structured event to the memory/events.jsonl log."""
        event = {
            "ts": time.time(),
            "type": event_type,
            "module": self.MODULE_ID,
            **data,
        }
        try:
            with open(self._events_log, "a") as f:
                f.write(json.dumps(event) + "\n")
        except Exception as exc:
            logger.debug("Event log write failed: %s", exc)

    # ------------------------------------------------------------------
    # Embedding fallback (zero-dependency)
    # ------------------------------------------------------------------

    def _hash_embed(self, text: str, dims: int = 384) -> np.ndarray:
        """Hash-based embedding fallback requiring only numpy + stdlib.

        Produces a deterministic, normalized vector from text.
        Lower quality than sentence-transformers but always available.
```

```
Use when your module has no dedicated embedder.
"""
import hashlib
rng_seed = int(hashlib.sha256(text.encode()).hexdigest(), 16) % (2**32)
rng = np.random.RandomState(rng_seed)
vec = rng.randn(dims).astype(np.float32)
norm = np.linalg.norm(vec)
return vec / norm if norm > 0 else vec
```