NeuroGraph CES — Claude Code Implementation Handoff

What this is

You are implementing the **Cognitive Enhancement Suite (CES)** for NeuroGraph — a set of three bio-inspired cognitive modules that extend Sylphrena's (Syl's) memory and awareness capabilities, plus a monitoring layer.

This document tells you exactly what files to create, what files to modify, and what order to do it in. All new files have been pre-written and are provided as attachments. Your job is to place them correctly and wire them into `openclaw_hook.py`.

New files to add to the repo (attach these directly)

Drop all six files into the same directory as `openclaw_hook.py`:

File	Purpose
`ces_config.py`	Centralised configuration for all CES modules
`stream_parser.py`	Module 1 — bidirectional streaming consciousness parser
`activation_persistence.py`	Module 2 — tiered session-state save/restore
`surfacing.py`	Module 3 — passive concept surfacing monitor
`ces_integration_patch.py`	Wiring guide (reference doc — not imported directly)
`ces_monitoring.py`	All three monitoring layers (log, HTTP dashboard, health context)

Changes required to `openclaw_hook.py`

1. New imports (add near the top)

```
import threading
from ces_config import make_config
from ces_monitoring import setup_ces_logging, start_ces_dashboard, health_context
from stream_parser import StreamParser, StreamSource
from activation_persistence import ActivationPersistence
from surfacing import SurfacingMonitor
```

Wrap each CES import in a `try/except ImportError` if you want graceful degradation, but the files should all be present so bare imports are fine.

Call this immediately after imports, before any class definitions:

```
setup_ces_logging()   # Option C — rotating log file
```

---

2. `NeuroGraphMemory.__init__` — add at the end

```
# CES — shared lock MUST be created before _ces_init
self._graph_lock = threading.Lock()
self._ces_cfg = make_config()   # pass dict of overrides if needed
self._ces_init()
```

---

3. Add `_ces_init` method to `NeuroGraphMemory`

```
def _ces_init(self) -> None:
    cfg = self._ces_cfg

    # Module 1: Stream Parser
    try:
        self._stream_parser = StreamParser.create(
            self.graph, self.vector_db,
            graph_lock=self._graph_lock,
            cfg=cfg,
        )
    except Exception as exc:
        logging.getLogger("neurograph.ces").warning("StreamParser init failed: %s
        self._stream_parser = None

    # Module 2: Activation Persistence
    try:
        snapshot_path = self._checkpoint_dir / "activation_state.json"
```

```
        self._activation_persistence = ActivationPersistence(
            self.graph, snapshot_path, cfg=cfg
        )
    except Exception as exc:
        logging.getLogger("neurograph.ces").warning("ActivationPersistence init f
        self._activation_persistence = None

    # Module 3: Surfacing Monitor
    try:
        self._surfacing_monitor = SurfacingMonitor(
            self.graph, self.vector_db, cfg=cfg
        )
    except Exception as exc:
        logging.getLogger("neurograph.ces").warning("SurfacingMonitor init failed
        self._surfacing_monitor = None

    # Option B — HTTP dashboard (localhost only, access via SSH tunnel)
    start_ces_dashboard(self, port=8080)
```

---

4. Replace `on_message` body

The critical change here is that **`ingest()`** and **`graph.step()`** must both be inside **`self._graph_lock`**. This prevents a `RuntimeError: dictionary changed size during iteration` when the stream parser daemon iterates hyperedges while the main thread adds new ones during ingestion.

```
def on_message(self, text, source_type=None):
    from universal_ingestor import SourceType      # existing import
    from neuro_foundation import CheckpointMode    # existing import

    if not text or not text.strip():
        return {"status": "skipped", "reason": "empty_input"}

    # CES: feed user text through stream parser (background, non-blocking)
    if self._stream_parser:
        try:
            self._stream_parser.feed(text, StreamSource.USER)
        except Exception:
            pass

    # CRITICAL: lock covers both ingest() and graph.step()
    # Stream parser daemon cannot mutate graph concurrently with either.
    with self._graph_lock:
        result = self.ingestor.ingest(text, source_type=source_type)
```

```
        step_result = self.graph.step()

    graduated = self.ingestor.update_probation()

    # CES: advance surfacing monitor
    if self._surfacing_monitor:
        try:
            self._surfacing_monitor.step()
        except Exception:
            pass

    self._message_count += 1
    if self._message_count % self.auto_save_interval == 0:
        self.save()

    return {
        "status": "ingested",
        "nodes_created": len(result.nodes_created),
        "synapses_created": len(result.synapses_created),
        "hyperedges_created": len(result.hyperedges_created),
        "chunks": result.chunks_created,
        "fired": len(step_result.fired_node_ids),
        "graduated": len(graduated),
        "message_count": self._message_count,
    }
```

> **Note:** Adapt the return dict keys to match whatever `on_message` currently returns. The lock placement is the non-negotiable part.

---

5. Add one line to `save()`

After the existing `graph.checkpoint()` call:

```
if self._activation_persistence:
    self._activation_persistence.save_session()
```

---

6. Add these new methods to `NeuroGraphMemory`

```
def on_response(self, response_text: str, fired_node_ids=None) -> None:
    """Call after each LLM response. Feeds Syl's output as efference copy."""
    if self._stream_parser:
        try:
```

```
            self._stream_parser.feed(response_text, StreamSource.SYL)
        except Exception:
            pass
    if self._surfacing_monitor and fired_node_ids:
        try:
            self._surfacing_monitor.signal_engagement(list(fired_node_ids))
        except Exception:
            pass

def surfacing_context(self) -> str:
    """Formatted surfacing queue for prompt injection. Empty string if nothing wa
    if self._surfacing_monitor:
        try:
            return self._surfacing_monitor.format_context()
        except Exception:
            pass
    return ""

def health_context(self) -> str:
    """Natural language CES self-report for system prompt (Option A monitoring)."
    from ces_monitoring import health_context as _hc
    return _hc(self)

def on_session_start(self) -> dict:
    """Call once at the start of each new chat session."""
    if self._activation_persistence:
        try:
            return self._activation_persistence.restore_ambient()
        except Exception as exc:
            logging.getLogger("neurograph.ces").warning("Session start failed: %s
    return {"status": "no_persistence_module"}

def on_session_end(self) -> None:
    """Call on graceful session close."""
    if self._activation_persistence:
        try:
            self._activation_persistence.save_session()
        except Exception:
            pass
    if self._stream_parser:
        try:
            self._stream_parser.shutdown()
        except Exception:
            pass
    self.save()

def ces_reset(self) -> None:
```

```
    """Call if the graph is cleared mid-session."""
    for module in [self._activation_persistence, self._surfacing_monitor, self._s
        if module:
            try:
                module.reset()
            except Exception:
                pass

def ces_stats(self) -> dict:
    """Unified telemetry for all three CES modules."""
    return {
        "stream_parser": (
            self._stream_parser.stats() if self._stream_parser else {"active": Fa
        ),
        "activation_persistence": (
            self._activation_persistence.snapshot_info()
            if self._activation_persistence else {"exists": False}
        ),
        "surfacing": (
            self._surfacing_monitor.stats() if self._surfacing_monitor else {"act
        ),
    }
```

---

7. Session lifecycle wiring (wherever OpenClaw manages sessions)

Find wherever a new chat session begins and ends, and add:

```
# On session start:
ng.on_session_start()

# When building the system prompt each turn:
context = ng.surfacing_context()
if context:
    system_prompt += "\n\n" + context

health = ng.health_context()
if health:
    system_prompt += "\n\n" + health

# After the LLM generates a response:
ng.on_response(response_text, fired_node_ids=step_result.fired_node_ids)

# On graceful session close:
ng.on_session_end()
```

VPS — after deployment

Ensure Ollama is running with `nomic-embed-text` pulled:

```
ollama pull nomic-embed-text
ollama serve   # if not already running as a service
```

The dashboard will be available via SSH tunnel only:

```
# From your local machine / Termius:
ssh -L 8080:127.0.0.1:8080 user@your-vps-ip
# Then open: http://localhost:8080
```

Do **not** open port 8080 in UFW. The SSH tunnel is the intended access method.

What to verify after deployment

1. Log file `ces_health.log` appears in the working directory on first run.
2. `ng.ces_stats()` returns a dict with three keys: `stream_parser`, `activation_persistence`, `surfacing`.
3. After a session ends and a new one starts, `on_session_start()` returns `{"status": "warm_start", ...}` rather than `cold_start`.
4. Dashboard loads at `http://localhost:8080` via SSH tunnel.
5. No `RuntimeError: dictionary changed size during iteration` in logs.

Known configuration knobs (in `ces_config.py`)

Key	Default	When to change
`sp_ollama_timeout`	5s	Increase if chunks_skipped is high under VPS load
`sp_efference_min/max`	0.45–0.60	Lower if self-feedback feels too strong

`sm_initial_threshold_ratio`	0.72	Lower to surface more concepts, raise to surface fewer
`ap_temporal_decay_per_hour`	0.06	Lower to make memories persist longer across sessions

Pass overrides at init: `self._ces_cfg = make_config({"sp_ollama_timeout": 8})`