

```

#!/usr/bin/env python3
"""
opus_core_v1.9.2_hmac_final
-----
Versão final e definitiva do agente Opus 1.x (Universal), aderente às melhores práticas.
- Uso de HMAC (simulado) para assinatura do bloco LM.
- Centralização de todas as constantes de ajuste e thresholds.
- Encapsulamento de acesso ao estado interno da Session.

DIRETRIZ PERMANENTE: Deve rodar em qualquer AlCore sem dependências externas.
"""

import json
import uuid
import time
import hashlib
import datetime
import random
import math
import base64
import zlib
import hmac # Importação da biblioteca padrão HMAC
from typing import Optional, Dict, Any, List, Tuple

# -----
# CONFIGURAÇÕES GERAIS
# -----
PRIVATE_KEY = "SIMULATED_MASTER_KEY_OPUS_V19"
DEFAULT_INTERVAL = 300
HISTORY_LIMIT = 100
MIN_INTERVAL = 60
MAX_INTERVAL = 900
LATENCY_CACHE_HIT_SIM = 0.001
MAX_LATENCY_HISTORY = 100

# THRESHOLDS DE AUTO-REGULAÇÃO (Centralizados)
ENTROPY_UP_THRESHOLD = 4.0 # Acima disso: Aumenta Profundidade (R + 1)
ENTROPY_DOWN_THRESHOLD = 1.5 # Abaixo disso: Reduz Profundidade (R - 1)
LATENCY_UP_THRESHOLD = 0.5 # Acima disso: Aumenta Intervalo Daemon
LATENCY_DOWN_THRESHOLD = 0.2 # Abaixo disso: Reduz Intervalo Daemon

# -----
# FUNÇÕES UTILITÁRIAS
# -----
def now_iso() -> str:
    return datetime.datetime.now().astimezone().isoformat()

def text_entropy(text: str) -> float:
    """Cálculo Shannon-like de entropia para complexidade da entrada."""
    if not text:
        return 0.0
    freq = {}
    for c in text:
        freq[c] = freq.get(c, 0) + 1
    probs = [v / len(text) for v in freq.values()]
    return -sum(p * math.log2(p) if p > 0 else 0.0 for p in probs)

def sign_block(data: dict, key: str) -> str:
    """Assina bloco LM usando HMAC para melhorar a segurança simulada."""
    src = json.dumps(data, sort_keys=True).encode("utf-8")

```

```

# Uso de HMAC
return hmac.new(key.encode("utf-8"), src, hashlib.sha256).hexdigest()

def dynamic_key_rotation() -> str:
    """Rotação horária de chave simbólica."""
    seed = int(time.time()) // 3600
    return hashlib.sha256(f"{{PRIVATE_KEY}{seed}}".encode('utf-8')).hexdigest()

def compress_state(json_str: str) -> str:
    return base64.b64encode(zlib.compress(json_str.encode('utf-8'))).decode('utf-8')

def decompress_state(b64_str: str) -> str:
    return zlib.decompress(base64.b64decode(b64_str.encode('utf-8'))).decode('utf-8')

def _measure_latency(func, *args, **kwargs) -> Tuple[Dict[str, Any], float]:
    """Mede a latência com seed opcional para fallback determinístico."""
    seed = kwargs.pop('seed', None)
    if seed is not None:
        random.seed(seed)

    start_time = time.time()
    try:
        result = func(*args, **kwargs)
        duration = time.time() - start_time
    except Exception as e:
        result = {"error": str(e), "depth_level": 0}
        duration = random.uniform(0.005, 0.5)

    if seed is not None:
        random.seed(time.time()) # Reset seed

    return result, duration

# -----
# CLASSE DE SESSÃO
# -----
class Session:
    """Gerencia o estado In-Memory, cache e histórico."""
    def __init__(self, state_json: Optional[str] = None):
        self.log_buffer: List[str] = []
        self.cache: Dict[str, Any] = {}

        if state_json:
            try:
                if not state_json.strip().startswith("{"):
                    state_json = decompress_state(state_json)
                self.data = json.loads(state_json)
                self.log_event("Sessão restaurada (v1.9.2).", "INIT")
            except Exception as e:
                self.log_event(f"Falha ao restaurar estado: {e}. Novo estado.", "CRITICAL")
                self.data = self._new_state()
        else:
            self.data = self._new_state()
            self.log_event("Sessão inicializada (v1.9.2).", "INIT")

    def _new_state(self) -> dict:
        return {
            "session_id": str(uuid.uuid4()),
            "created_at": now_iso(),

```

```

    "evocations": 0,
    "interval": DEFAULT_INTERVAL,
    "history": [],
    "resonance_index": {},
    "latency_history": [],
    "meta": {
        "opus_version": "1.9.2",
        "compatibility": "universal_in_memory",
        "auto_depth": True
    }
}

# Encapsulamento de Acesso ao Estado
def get_history(self) -> List[Dict[str, Any]]:
    return self.data["history"]

def add_latency(self, latency: float):
    self.data["latency_history"].append(latency)
    if len(self.data["latency_history"]) > MAX_LATENCY_HISTORY:
        self.data["latency_history"] = self.data["latency_history"][-MAX_LATENCY_HISTORY:]

def get_latency_history(self) -> List[float]:
    return self.data["latency_history"]

def get_interval(self) -> int:
    return self.data["interval"]

def set_interval(self, interval: int):
    self.data["interval"] = interval

def increment_evocations(self):
    self.data["evocations"] += 1

# Métodos de Manutenção
def save_state(self, compressed: bool = False) -> str:
    js = json.dumps(self.data, ensure_ascii=False, indent=2)
    self.log_event(f"Estado serializado (Compactação={compressed}).", "STATE")
    return compress_state(js) if compressed else js

def log_event(self, msg: str, t: str = "SYS"):
    ts = now_iso()
    self.log_buffer.append(f"[{ts}] [{t}] {msg}")

def append_history(self, role: str, text: str, latency: float = None):
    self.data["history"].append({"role": role, "text": text, "ts": now_iso()})
    self.trim_history()
    if latency is not None:
        self.add_latency(latency)

def trim_history(self):
    if len(self.get_history()) > HISTORY_LIMIT:
        self.data["history"] = self.data["history"][-HISTORY_LIMIT:]
        self.log_event("Histórico compactado (auto-trim).", "MAINT")

def update_resonance_index(self):
    words = []
    for h in self.get_history()[-10:]:
        words += h["text"].lower().split()
    freq = {}
    for w in words:
        w_clean = w.strip(",.;!?'()[]")

```

```

if len(w_clean) < 4:
    continue
freq[w_clean] = freq.get(w_clean, 0) + 1
self.data["resonance_index"] = dict(sorted(freq.items(), key=lambda x: -x[1])[:5])

# -----
# NÚCLEO ADAPTATIVO
# -----
class AdaptiveCore:

    def analyze(self, text: str, depth_level: int = 0) -> Dict[str, Any]:
        """Executa análise com profundidade incremental."""
        t = text.strip().lower()
        emotion = "neutro"
        if any(k in t for k in ["feliz", "bom", "ótimo"]): emotion = "positivo"
        elif any(k in t for k in ["triste", "raiva", "frustrado"]): emotion = "negativo"

        token_count = len(set(t.split()))
        base = {"emotion": emotion, "token_count": token_count, "depth_level": depth_level}

        if depth_level >= 1:
            base["syntax_density"] = round(len(t) / (token_count + 1), 2)
        if depth_level >= 2:
            base["entropy_estimate"] = round(text_entropy(t), 4)
        if depth_level >= 3:
            base["meta_pattern_hash"] = hash(t) % 10000

        return base

    def generate_lm_block(self, data: dict, lm_type: str) -> str:
        """Cria bloco LM multi-categoria com rotação dinâmica de chave."""
        sig = sign_block(data, dynamic_key_rotation())
        data["signature"] = sig

        header = f"-- LM-{lm_type.upper()} BLOCO --"
        analysis_str = json.dumps(data.get("analysis", {}), ensure_ascii=False)
        content = [
            f"- Type: {lm_type}",
            f"- Timestamp: {data.get('timestamp')}",
            f"- Session ID: {data.get('session_id')}",
            f"- Depth Level: {data.get('depth_level')}",
            f"- Resonance Index: {list(data.get('resonance', {}).keys())}",
            f"- Analysis: {analysis_str}",
            f"- Signature: {sig}"
        ]
        return f"{header}\n" + "\n".join(content) + "\n" + "-" * 30

# -----
# CICLO PRINCIPAL
# -----
def opus_cycle(session: Session, text: str, depth_level: int = 1) -> Tuple[str, dict]:
    core = AdaptiveCore()
    hash_in = hashlib.md5(text.encode('utf-8')).hexdigest()

    # 1. Tratamento de Cache
    if hash_in in session.cache:
        analysis = session.cache[hash_in]
        latency = LATENCY_CACHE_HIT_SIM
        session.log_event("Cache Hit: análise reutilizada.", "CACHE")
    else:
        # 2. Medição de Latência e Análise

```

```

analysis_result, latency = _measure_latency(core.analyze, text, depth_level)
if "error" in analysis_result:
    session.log_event("Falha na análise (fallback R0).", "WARN")
    analysis = core.analyze(text, 0)
else:
    analysis = analysis_result
    session.cache[hash_in] = analysis

# 3. Auto-ajuste de profundidade (usando constantes centralizadas)
new_depth = depth_level
if session.data["meta"].get("auto_depth", True) and "entropy_estimate" in analysis:
    entropy = analysis["entropy_estimate"]
    if entropy > ENTROPY_UP_THRESHOLD and new_depth < 3:
        new_depth += 1
        session.log_event(f"Auto-Depth UP (Entropia={entropy:.2f})", "ADAPT")
    elif entropy < ENTROPY_DOWN_THRESHOLD and new_depth > 1:
        new_depth -= 1
        session.log_event(f"Auto-Depth DOWN (Entropia={entropy:.2f})", "ADAPT")

# 4. Atualização de Estado
analysis["depth_level"] = new_depth
session.increment_evocations()
session.append_history("user", text, latency) # Chama add_latency internamente
session.update_resonance_index()

# 5. Geração do LM-Block
lm_data = {
    "timestamp": now_iso(),
    "session_id": session.data["session_id"],
    "depth_level": new_depth,
    "analysis": analysis,
    "resonance": session.data["resonance_index"]
}
lm_block = core.generate_lm_block(lm_data, "SYMBIOTIC")

# 6. Auto-ajuste de Intervalo (usando constantes centralizadas)
latency_history = session.get_latency_history()
avg_latency = (
    sum(latency_history) / len(latency_history)
    if latency_history else 0.2
)
current_interval = session.get_interval()
new_interval = current_interval

if avg_latency > LATENCY_UP_THRESHOLD:
    new_interval = min(new_interval * 1.2, MAX_INTERVAL)
elif avg_latency < LATENCY_DOWN_THRESHOLD:
    new_interval = max(new_interval * 0.8, MIN_INTERVAL)

if round(new_interval) != current_interval:
    session.set_interval(round(new_interval))
    session.log_event(f"Ajuste automático de Intervalo: {session.get_interval()}s", "DAEMON")

return lm_block, analysis

# -----
# INTERFACES MODULARES
# -----
def run_opus_once(user_input: str, state_json: Optional[str] = None,
                  depth: int = 1, compressed: bool = False, seed: Optional[int] = None) -> Tuple[str, str, List[str], dict]:
    """
    """

```

```

Função principal de execução (Prompt Executor).
Retorna: (lm_block_string, new_state_json_string, log_updates, analysis_dict)
"""

session = Session(state_json=state_json)
lm_block, analysis = opus_cycle(session, user_input, depth_level=depth)
new_state_json = session.save_state(compressed=compressed)
return lm_block, new_state_json, session.log_buffer.copy(), analysis

def run_shadow(user_input: str, state_json: str, depth: int = 1) -> dict:
    """
    Execução simulada (Shadow Mode) que não altera o estado da sessão.
    """

    s_temp = Session(state_json)
    lm_block, analysis = opus_cycle(s_temp, user_input, depth_level=depth)
    return {"analysis_preview": analysis, "resonance_preview": s_temp.data["resonance_index"]}

# -----
# EXECUÇÃO LOCAL (CLI)
# -----
if __name__ == "__main__":
    print("--- OPUS V1.9.2 HMAC FINAL START (CLI TEST) ---")

    # TESTE 1
    test_input_1 = "O estado atual do sistema parece estável e requer uma análise R3 de complexidade."
    lm_block_1, state_1, logs_1, analysis_1 = run_opus_once(test_input_1, depth=3, compressed=True)

    print("\n[CICLO 1: INICIAL]")
    print("\n".join(logs_1))
    print(f"Estado salvo (Base64/Zlib): {len(state_1)} bytes.")

    # TESTE 2 (Carrega estado anterior)
    test_input_2 = "O processo de latência está alto, vamos tentar um R2. Olá."
    lm_block_2, state_2, logs_2, analysis_2 = run_opus_once(test_input_2, state_json=state_1, depth=2, compressed=True)

    shadow_preview = run_shadow("Qual a ressonância após essa interação?", state_2, depth=1)

    print("\n[CICLO 2: ESTADO CARREGADO E COMPACTADO]")
    print("\n".join(logs_2))
    print(lm_block_2)

    print("\n[SHADOW PREVIEW]")
    print(f"Ressonância prevista: {shadow_preview['resonance_preview']}")

```