"""

═══════════════════════════════════════════════════════════════════════════════

TACTIK ALGORITHM 5.3 PREMIUM EDITION

Strategic Intelligence System with AVDA Validation

═══════════════════════════════════════════════════════════════════════════════

Author: Santiago Cañas (TACTIK AI)

Version: 5.3 Premium

Date: October 2025

ENHANCED FEATURES:

✓ AVDA Validation System (Avatar Validation Deviation & Accuracy)

✓ Enhanced Empathy Pause with Multi-Factor Triggers

✓ Advanced Multi-Avatar Orchestration (up to 4 simultaneous)

✓ DNA Builder with Ground Truth Sources Tracking

✓ Transparency Card Auto-Generation

✓ Hybrid IA-Human Session Support

✓ Enterprise-Grade Monitoring & Observability

APPLICATIONS (VERTICAL AGNOSTIC):

✓ Diplomacy & Trade Negotiations

✓ Corporate Strategy & M&A

✓ Education & Training (Socratic Method)

✓ Legal Practice & Litigation Prep

✓ Healthcare (Patient Communication Training)

✓ Government Policy Simulation

✓ Sales & Business Development

✓ Crisis Management & PR

BENCHMARKS:

- Target Latency: <500ms

- Target Accuracy: >85%

- AVDA Score: >75% (High Fidelity)

- Concurrent Sessions: 10,000+

- Uptime SLA: 99.9%

═══════════════════════════════════════════════════════════════════════════════

"""

import asyncio

import logging

import time

import uuid

import json

from dataclasses import dataclass, field, asdict

from typing import Dict, List, Optional, Tuple, Any, Callable

from enum import Enum

from contextlib import asynccontextmanager

from collections import deque

import numpy as np

# ═══════════════════════════════════════════════════════════════════════════════

# CONFIGURATION & CONSTANTS

# ═══════════════════════════════════════════════════════════════════════════════

class SessionPhase(Enum):

"""Fases de sesión conversacional"""

INIT = "init"

INFO\_WAIT = "info\_wait"

GENERATE = "generate"

BACKFLOW = "backflow"

EMPATHY\_PAUSE = "empathy\_pause"

END = "end"

class AvatarFidelity(Enum):

"""Clasificación fidelidad avatar según AVDA Score"""

VERY\_HIGH = "VERY\_HIGH\_FIDELITY" # 90-100%

HIGH = "HIGH\_FIDELITY" # 75-89%

MEDIUM = "MEDIUM\_FIDELITY" # 60-74%

LOW = "LOW\_FIDELITY" # 45-59%

UNRELIABLE = "UNRELIABLE" # <45%

# Configuración Gating con Histéresis

TAU0 = 0.52

TRUST\_SLOPE = -0.12

SENS\_BONUS = 0.08

TAU\_RAISE\_AFTER\_INTERRUPT = 0.05

# Configuración Backflow

BACKFLOW\_THRESHOLDS = {

"drift": 0.40,

"uncertainty": 0.65,

"coverage": 0.40,

"contradiction": 0.50

}

# Configuración Empathy Pause Enhanced

EMPATHY\_PAUSE\_CONFIG = {

"min\_turns": 3,

"cooldown\_turns": 6,

"trigger\_threshold": 2,

"complexity\_weight": 0.30,

"success\_rate\_weight": 0.25,

"error\_weight": 0.25,

"sentiment\_weight": 0.20

}

# Configuración Multi-Avatar

MAX\_AVATARS = 4

MAX\_HUMANS = 3

# Configuración AVDA

AVDA\_CONFIG = {

"min\_sources\_tier\_1\_2": 3,

"confidence\_level": 0.95,

"recency\_penalty\_months": 24,

"cross\_verification\_bonus": 0.10

}

# Configuración EMA

LAMBDA\_EMA = 0.30

# ═══════════════════════════════════════════════════════════════════════════════

# DATA STRUCTURES

# ═══════════════════════════════════════════════════════════════════════════════

@dataclass

class GroundTruthSource:

"""Fuente de información verificada para DNA avatar"""

source\_id: str

tier: str # tier\_1\_primary, tier\_2\_secondary, tier\_3\_tertiary, tier\_4\_inferred

description: str

url: Optional[str] = None

date: str = ""

cross\_verified: bool = False

confidence: float = 0.0

def get\_weight(self) -> float:

weights = {

"tier\_1\_primary": 1.0,

"tier\_2\_secondary": 0.85,

"tier\_3\_tertiary": 0.60,

"tier\_4\_inferred": 0.40

}

return weights.get(self.tier, 0.40)

@dataclass

class AvatarDNA:

"""DNA cognitivo avatar con fuentes verificadas"""

avatar\_id: str

avatar\_name: str

influences: Dict[str, Any]

thoughts: Dict[str, Any]

environment: str

behavioral\_pattern: Dict[str, Any]

decision\_style: Dict[str, Any]

communication\_style: Dict[str, Any]

priorities: Dict[str, Any]

constraints: Dict[str, Any]

language: str = "es"

sources: List[GroundTruthSource] = field(default\_factory=list)

def calculate\_source\_coverage(self) -> Tuple[float, Dict]:

components = [

"influences", "thoughts", "behavioral\_pattern",

"decision\_style", "communication\_style", "priorities"

]

coverage\_scores = {}

for component in components:

data = getattr(self, component, {})

verified = len(data.get("verified", []))

inferred = len(data.get("inferred", []))

total = verified + inferred

if total == 0:

coverage = 0.0

else:

verified\_weight = sum([

s.get\_weight() for s in self.sources

if s.source\_id in [v.get("source\_id", "") for v in data.get("verified", [])]

])

coverage = verified\_weight / total if total > 0 else 0.0

coverage\_scores[component] = coverage

total\_coverage = (

0.15 \* coverage\_scores.get("influences", 0) +

0.20 \* coverage\_scores.get("thoughts", 0) +

0.15 \* coverage\_scores.get("communication\_style", 0) +

0.30 \* coverage\_scores.get("behavioral\_pattern", 0) +

0.20 \* coverage\_scores.get("decision\_style", 0)

)

return total\_coverage, coverage\_scores

@dataclass

class AVDAMetrics:

"""Métricas Avatar Validation Deviation & Accuracy"""

accuracy: float

confidence\_interval: Tuple[float, float]

source\_coverage: float

drift\_risk: float

ground\_truth\_quality: float

avda\_score: float

classification: AvatarFidelity

limitations: List[str] = field(default\_factory=list)

source\_breakdown: Dict = field(default\_factory=dict)

def to\_percentage(self) -> Dict:

return {

"accuracy": round(self.accuracy \* 100, 1),

"confidence\_interval": [round(self.confidence\_interval[0] \* 100, 1),

round(self.confidence\_interval[1] \* 100, 1)],

"source\_coverage": round(self.source\_coverage \* 100, 1),

"drift\_risk": round(self.drift\_risk \* 100, 1),

"ground\_truth\_quality": round(self.ground\_truth\_quality \* 100, 1),

"avda\_score": round(self.avda\_score \* 100, 1),

"classification": self.classification.value

}

@dataclass

class SessionMetrics:

"""Métricas sesión conversacional"""

EIS: float = 0.0

HCA: float = 0.0

DNA: float = 0.0

TACTIK: float = 0.0

trust: float = 0.5

success\_rate: float = 1.0

recent\_errors: int = 0

avg\_latency: float = 0.0

backflow\_count: int = 0

empathy\_pause\_count: int = 0

gate\_interrupt\_count: int = 0

@dataclass

class SessionContext:

"""Contexto completo de sesión"""

session\_id: str

profile: str

active\_avatar: Optional[str] = None

preselected\_avatars: List[str] = field(default\_factory=list)

invited\_humans: List[str] = field(default\_factory=list)

phase: SessionPhase = SessionPhase.INIT

turn\_count: int = 0

last\_interrupt\_turn: Optional[int] = None

transcript: List[Tuple[str, str]] = field(default\_factory=list)

metrics: SessionMetrics = field(default\_factory=SessionMetrics)

context\_history: deque = field(default\_factory=lambda: deque(maxlen=50))

def can\_interrupt(self) -> bool:

if self.last\_interrupt\_turn is None:

return True

return (self.turn\_count - self.last\_interrupt\_turn) >= EMPATHY\_PAUSE\_CONFIG["cooldown\_turns"]

def note\_backflow(self) -> None:

self.metrics.backflow\_count += 1

def note\_empathy\_pause(self) -> None:

self.metrics.empathy\_pause\_count += 1

self.last\_interrupt\_turn = self.turn\_count

def note\_gate\_interrupt(self) -> None:

self.metrics.gate\_interrupt\_count += 1

self.last\_interrupt\_turn = self.turn\_count

# ═══════════════════════════════════════════════════════════════════════════════

# CORE ENGINE

# ═══════════════════════════════════════════════════════════════════════════════

class TactikEngine5\_3:

"""Motor principal TACTIK 5.3 Premium Edition"""

def \_\_init\_\_(self, config: Optional[Dict] = None):

self.config = config or {}

self.logger = self.\_setup\_logger()

self.avatars\_dna: Dict[str, AvatarDNA] = {}

self.sessions: Dict[str, SessionContext] = {}

def \_setup\_logger(self) -> logging.Logger:

logger = logging.getLogger("tactik\_5.3")

logger.setLevel(logging.INFO)

handler = logging.StreamHandler()

formatter = logging.Formatter(

'{"timestamp":"%(asctime)s","level":"%(levelname)s","message":"%(message)s"}'

)

handler.setFormatter(formatter)

logger.addHandler(handler)

return logger

# ═══════════════════════════════════════════════════════════════════════════

# DNA BUILDER

# ═══════════════════════════════════════════════════════════════════════════

def build\_avatar\_dna(self, avatar\_id: str, avatar\_name: str,

dna\_components: Dict, sources: List[GroundTruthSource]) -> AvatarDNA:

"""Construye DNA avatar con fuentes verificadas"""

avatar\_dna = AvatarDNA(

avatar\_id=avatar\_id, avatar\_name=avatar\_name,

influences=dna\_components.get("influences", {}),

thoughts=dna\_components.get("thoughts", {}),

environment=dna\_components.get("environment", ""),

behavioral\_pattern=dna\_components.get("behavioral\_pattern", {}),

decision\_style=dna\_components.get("decision\_style", {}),

communication\_style=dna\_components.get("communication\_style", {}),

priorities=dna\_components.get("priorities", {}),

constraints=dna\_components.get("constraints", {}),

language=dna\_components.get("language", "es"),

sources=sources

)

self.avatars\_dna[avatar\_id] = avatar\_dna

self.logger.info(f"Avatar DNA built: {avatar\_id}", extra={"sources\_count": len(sources)})

return avatar\_dna

# ═══════════════════════════════════════════════════════════════════════════

# COGNITIVE DECONSTRUCTION

# ═══════════════════════════════════════════════════════════════════════════

async def \_cognitive\_deconstruction(self, user\_input: str, session: SessionContext) -> Dict[str, Any]:

"""Análisis multi-dimensional del input usuario"""

return {

"intent": self.\_classify\_intent(user\_input),

"entities": self.\_extract\_entities(user\_input),

"sentiment": self.\_analyze\_sentiment(user\_input),

"confidence": 0.85,

"intent\_clarity": 0.90,

"complexity": self.\_assess\_complexity(user\_input)

}

def \_classify\_intent(self, text: str) -> str:

if "?" in text: return "question"

elif any(w in text.lower() for w in ["propongo", "sugiero", "ofrezco"]): return "proposal"

elif any(w in text.lower() for w in ["confirmo", "certifico", "valido"]): return "validation"

else: return "statement"

def \_extract\_entities(self, text: str) -> List[str]:

entities = []

keywords = ["USD", "Ecuador", "certificación", "proyecto"]

for keyword in keywords:

if keyword in text: entities.append(keyword)

return entities

def \_analyze\_sentiment(self, text: str) -> str:

negative = ["problema", "rechazo", "imposible", "no"]

positive = ["excelente", "confirmo", "sí", "acepto"]

neg\_count = sum(1 for w in negative if w in text.lower())

pos\_count = sum(1 for w in positive if w in text.lower())

if pos\_count > neg\_count: return "positive"

elif neg\_count > pos\_count: return "negative"

else: return "neutral"

def \_assess\_complexity(self, text: str) -> str:

word\_count = len(text.split())

if word\_count > 50: return "high"

elif word\_count > 20: return "medium"

else: return "low"

# ═══════════════════════════════════════════════════════════════════════════

# GATING WITH HYSTERESIS

# ═══════════════════════════════════════════════════════════════════════════

def gate\_decision(self, context: Dict[str, Any], session: SessionContext) -> Dict[str, bool]:

"""Sistema de compuertas con histéresis"""

risk = context.get("risk", 0.0)

uncertainty = context.get("uncertainty", 0.0)

drift = context.get("drift", 0.0)

sensitivity = context.get("sensitivity", 0.0)

latency\_pressure = context.get("latency\_pressure", 0.0)

trust = session.metrics.trust

score = 0.33\*risk + 0.18\*uncertainty + 0.16\*drift + 0.18\*sensitivity + 0.15\*latency\_pressure

tau = TAU0 + TRUST\_SLOPE\*trust

if sensitivity > 0.7: tau += SENS\_BONUS

if session.last\_interrupt\_turn is not None:

turns\_since = session.turn\_count - session.last\_interrupt\_turn

if turns\_since < EMPATHY\_PAUSE\_CONFIG["cooldown\_turns"]: tau += TAU\_RAISE\_AFTER\_INTERRUPT

interrupt = score >= tau and session.can\_interrupt()

result = {"approved": not interrupt, "interrupt": interrupt, "score": score, "threshold": tau}

if interrupt:

result["reason"] = f"Risk/uncertainty/drift threshold exceeded (score={score:.3f} >= tau={tau:.3f})"

session.note\_gate\_interrupt()

return result

# ═══════════════════════════════════════════════════════════════════════════

# BACKFLOW DETECTION

# ═══════════════════════════════════════════════════════════════════════════

def detect\_backflow(self, context: Dict, session: SessionContext) -> Dict[str, Any]:

"""Detecta 4 tipos problemas: drift, uncertainty, coverage gap, contradiction"""

drift = context.get("drift", 0.0)

uncertainty = context.get("uncertainty", 0.0)

coverage = context.get("coverage", 1.0)

contradiction = context.get("contradiction", 0.0)

triggers = []

if drift > BACKFLOW\_THRESHOLDS["drift"]:

triggers.append(("drift", drift, "Topic drift detected"))

if uncertainty > BACKFLOW\_THRESHOLDS["uncertainty"]:

triggers.append(("uncertainty", uncertainty, "High uncertainty detected"))

if coverage < BACKFLOW\_THRESHOLDS["coverage"]:

triggers.append(("coverage", 1-coverage, "Coverage gap detected"))

if contradiction > BACKFLOW\_THRESHOLDS["contradiction"]:

triggers.append(("contradiction", contradiction, "Contradiction detected"))

if triggers:

session.note\_backflow()

return {

"activated": True,

"triggers": triggers,

"correction": "Let me refocus to address your core question accurately."

}

return {"activated": False}

# ═══════════════════════════════════════════════════════════════════════════

# ENHANCED EMPATHY PAUSE

# ═══════════════════════════════════════════════════════════════════════════

async def check\_empathy\_pause(self, session: SessionContext, context: Dict[str, Any],

cognitive\_analysis: Dict) -> Dict[str, Any]:

"""Empathy Pause Enhanced con triggers multi-factor"""

if not session.can\_interrupt(): return {"activated": False}

if session.turn\_count < EMPATHY\_PAUSE\_CONFIG["min\_turns"]: return {"activated": False}

conditions = {

"high\_complexity": cognitive\_analysis.get("complexity") == "high",

"low\_success\_rate": session.metrics.success\_rate < 0.5,

"many\_errors": session.metrics.recent\_errors > 2,

"negative\_sentiment": cognitive\_analysis.get("sentiment") == "negative"

}

conditions\_met = sum(conditions.values())

if conditions\_met >= EMPATHY\_PAUSE\_CONFIG["trigger\_threshold"]:

session.note\_empathy\_pause()

return {

"activated": True,

"conditions": conditions,

"response": "I want to ensure accuracy. Let me pause and reconsider the best approach."

}

return {"activated": False, "conditions\_met": conditions\_met}

# ═══════════════════════════════════════════════════════════════════════════

# EMA METRICS

# ═══════════════════════════════════════════════════════════════════════════

def calculate\_ema\_metrics(self, context: Dict[str, Any], session: SessionContext) -> SessionMetrics:

"""Calcula métricas EMA: EIS, HCA, DNA, TACTIK"""

H = context.get("uncertainty", 0.0)

P = context.get("precision", 0.8)

R = context.get("relevance", 0.85)

L = context.get("latency", 0.2)

E = context.get("error", 0.1)

S = context.get("sensitivity", 0.75)

C = context.get("coherence", 0.85)

V = context.get("variability", 0.70)

A = context.get("adaptability", 0.80)

drift = context.get("drift", 0.15)

historical\_acc = context.get("historical\_accuracy", 0.90)

eis\_now = 0.35\*(1-H) + 0.25\*P + 0.20\*R + 0.10\*(1-L) + 0.10\*(1-E)

hca\_now = 0.35\*S + 0.30\*C + 0.20\*V + 0.15\*A

s\_mean = (S+C+V+A)/4.0

dna\_now = s\_mean \* (1-drift) \* historical\_acc

lambda\_ = LAMBDA\_EMA

session.metrics.EIS = lambda\_\*eis\_now + (1-lambda\_)\*session.metrics.EIS

session.metrics.HCA = lambda\_\*hca\_now + (1-lambda\_)\*session.metrics.HCA

session.metrics.DNA = lambda\_\*dna\_now + (1-lambda\_)\*session.metrics.DNA

session.metrics.TACTIK = 10 \* (0.4\*session.metrics.EIS + 0.3\*session.metrics.HCA + 0.3\*session.metrics.DNA)

return session.metrics

# ═══════════════════════════════════════════════════════════════════════════

# AVDA VALIDATION

# ═══════════════════════════════════════════════════════════════════════════

def calculate\_avda\_score(self, avatar\_id: str, session: SessionContext) -> AVDAMetrics:

"""Calcula AVDA (Avatar Validation Deviation & Accuracy)"""

avatar\_dna = self.avatars\_dna.get(avatar\_id)

if not avatar\_dna: raise ValueError(f"Avatar DNA not found: {avatar\_id}")

gt\_quality = self.\_validate\_ground\_truth\_quality(avatar\_dna.sources)

behavior\_fidelity = 0.85 # Placeholder

source\_coverage, coverage\_breakdown = avatar\_dna.calculate\_source\_coverage()

drift\_risk = 0.25 if session.metrics.DNA < 0.70 else 0.10

ci\_lower, ci\_upper = self.\_calculate\_confidence\_interval(behavior\_fidelity, source\_coverage, len(avatar\_dna.sources))

avda\_final = 0.40\*behavior\_fidelity\*gt\_quality + 0.30\*source\_coverage + 0.20\*(1-drift\_risk) + 0.10\*gt\_quality

classification = self.\_classify\_fidelity(avda\_final)

limitations = []

if len([s for s in avatar\_dna.sources if s.tier=="tier\_1\_primary"]) < 2:

limitations.append("Limited primary sources")

return AVDAMetrics(

accuracy=behavior\_fidelity, confidence\_interval=(ci\_lower, ci\_upper),

source\_coverage=source\_coverage, drift\_risk=drift\_risk,

ground\_truth\_quality=gt\_quality, avda\_score=avda\_final,

classification=classification, limitations=limitations,

source\_breakdown=coverage\_breakdown

)

def \_validate\_ground\_truth\_quality(self, sources: List[GroundTruthSource]) -> float:

if not sources: return 0.0

weighted\_score = sum(s.get\_weight() for s in sources)

tier\_1\_2\_count = sum(1 for s in sources if s.tier in ["tier\_1\_primary", "tier\_2\_secondary"])

diversity\_penalty = 1.0 if tier\_1\_2\_count >= AVDA\_CONFIG["min\_sources\_tier\_1\_2"] else 0.70

return min((weighted\_score / len(sources)) \* diversity\_penalty, 1.0)

def \_calculate\_confidence\_interval(self, behavior\_score: float, coverage\_score: float, sample\_size: int) -> Tuple[float, float]:

avg\_score = (behavior\_score + coverage\_score) / 2

penalty\_factor = 2.0 if sample\_size < 3 else (1.5 if sample\_size < 10 else 1.0)

standard\_error = np.sqrt(avg\_score \* (1-avg\_score) / sample\_size) \* penalty\_factor

z\_score = 1.96

return max(0.0, avg\_score - z\_score\*standard\_error), min(1.0, avg\_score + z\_score\*standard\_error)

def \_classify\_fidelity(self, avda\_score: float) -> AvatarFidelity:

if avda\_score >= 0.90: return AvatarFidelity.VERY\_HIGH

elif avda\_score >= 0.75: return AvatarFidelity.HIGH

elif avda\_score >= 0.60: return AvatarFidelity.MEDIUM

elif avda\_score >= 0.45: return AvatarFidelity.LOW

else: return AvatarFidelity.UNRELIABLE

# ═══════════════════════════════════════════════════════════════════════════

# MULTI-AVATAR ORCHESTRATION

# ═══════════════════════════════════════════════════════════════════════════

def create\_session(self, profile: str, avatars: List[str], session\_id: Optional[str] = None) -> SessionContext:

"""Crea nueva sesión multi-avatar"""

if not avatars or len(avatars) > MAX\_AVATARS:

raise ValueError(f"Must select 1-{MAX\_AVATARS} avatars")

for avatar\_id in avatars:

if avatar\_id not in self.avatars\_dna:

raise ValueError(f"Avatar DNA not found: {avatar\_id}")

session\_id = session\_id or str(uuid.uuid4())

session = SessionContext(

session\_id=session\_id, profile=profile,

preselected\_avatars=avatars, active\_avatar=avatars[0]

)

self.sessions[session\_id] = session

self.logger.info("Session created", extra={"session\_id": session\_id, "avatars": avatars})

return session

def switch\_avatar(self, session\_id: str, avatar\_id: str) -> None:

"""Cambia avatar activo"""

session = self.sessions.get(session\_id)

if not session: raise ValueError(f"Session not found: {session\_id}")

if avatar\_id not in session.preselected\_avatars: raise ValueError(f"Avatar not preselected: {avatar\_id}")

session.active\_avatar = avatar\_id

self.logger.info("Avatar switched", extra={"session\_id": session\_id, "new\_avatar": avatar\_id})

def invite\_human(self, session\_id: str, human\_id: str) -> None:

"""Invita humano real a sesión"""

session = self.sessions.get(session\_id)

if not session: raise ValueError(f"Session not found: {session\_id}")

if len(session.invited\_humans) >= MAX\_HUMANS: raise ValueError(f"Maximum {MAX\_HUMANS} humans allowed")

if human\_id not in session.invited\_humans:

session.invited\_humans.append(human\_id)

self.logger.info("Human invited", extra={"session\_id": session\_id, "human\_id": human\_id})

def post\_human\_turn(self, session\_id: str, human\_id: str, text: str) -> Dict[str, Any]:

"""Registra turno de humano invitado"""

session = self.sessions.get(session\_id)

if not session: raise ValueError(f"Session not found: {session\_id}")

if human\_id not in session.invited\_humans: raise ValueError(f"Human not invited: {human\_id}")

session.transcript.append((f"human:{human\_id}", text))

session.turn\_count += 1

session.metrics.trust = min(session.metrics.trust + 0.05, 1.0)

return {"ok": True, "turn": session.turn\_count}

# ═══════════════════════════════════════════════════════════════════════════

# MAIN CONVERSATION LOOP

# ═══════════════════════════════════════════════════════════════════════════

async def process\_turn(self, session\_id: str, user\_input: str, context: Optional[Dict[str, Any]] = None) -> Dict[str, Any]:

"""Procesa turno conversacional completo"""

session = self.sessions.get(session\_id)

if not session: raise ValueError(f"Session not found: {session\_id}")

context = context or {}

session.turn\_count += 1

cognitive\_analysis = await self.\_cognitive\_deconstruction(user\_input, session)

full\_context = {\*\*context, \*\*cognitive\_analysis}

gate\_result = self.gate\_decision(full\_context, session)

if gate\_result["interrupt"]:

return {"type": "gate\_interrupt", "reason": gate\_result["reason"], "turn": session.turn\_count}

backflow\_result = self.detect\_backflow(full\_context, session)

if backflow\_result["activated"]:

return {"type": "backflow\_correction", "correction": backflow\_result["correction"], "turn": session.turn\_count}

empathy\_result = await self.check\_empathy\_pause(session, full\_context, cognitive\_analysis)

if empathy\_result["activated"]:

return {"type": "empathy\_pause", "response": empathy\_result["response"], "turn": session.turn\_count}

avatar\_response = f"[{session.active\_avatar}] Response to your {cognitive\_analysis['intent']}"

updated\_metrics = self.calculate\_ema\_metrics(full\_context, session)

session.transcript.append(("user", user\_input))

session.transcript.append((f"avatar:{session.active\_avatar}", avatar\_response))

session.metrics.success\_rate = 1.0 - (

(session.metrics.backflow\_count + session.metrics.empathy\_pause\_count + session.metrics.gate\_interrupt\_count)

/ session.turn\_count

)

return {

"type": "response", "avatar": session.active\_avatar, "response": avatar\_response,

"metrics": {"EIS": round(updated\_metrics.EIS,3), "HCA": round(updated\_metrics.HCA,3),

"DNA": round(updated\_metrics.DNA,3), "TACTIK": round(updated\_metrics.TACTIK,2)},

"turn": session.turn\_count, "session\_id": session\_id

}

# ═══════════════════════════════════════════════════════════════════════════

# REPORTING

# ═══════════════════════════════════════════════════════════════════════════

def generate\_tactik\_advisor\_report(self, session\_id: str) -> Dict[str, Any]:

"""Genera informe TACTIK Advisor completo"""

session = self.sessions.get(session\_id)

if not session: raise ValueError(f"Session not found: {session\_id}")

avda\_scores = {}

for avatar\_id in session.preselected\_avatars:

avda = self.calculate\_avda\_score(avatar\_id, session)

avda\_scores[avatar\_id] = avda.to\_percentage()

return {

"session\_id": session\_id, "total\_turns": session.turn\_count,

"avda\_scores": avda\_scores,

"metrics": {

"EIS\_final": round(session.metrics.EIS, 3),

"HCA\_final": round(session.metrics.HCA, 3),

"DNA\_final": round(session.metrics.DNA, 3),

"TACTIK\_final": round(session.metrics.TACTIK, 2),

"success\_rate": round(session.metrics.success\_rate, 3)

}

}

def generate\_transparency\_card(self, session\_id: str, avatar\_id: str) -> str:

"""Genera Transparency Card completa para avatar"""

session = self.sessions.get(session\_id)

if not session: raise ValueError(f"Session not found: {session\_id}")

avatar\_dna = self.avatars\_dna.get(avatar\_id)

if not avatar\_dna: raise ValueError(f"Avatar DNA not found: {avatar\_id}")

avda = self.calculate\_avda\_score(avatar\_id, session)

avda\_pct = avda.to\_percentage()

card = f"""

═══════════════════════════════════════════════════════════════════

TACTIK SIMULATION TRANSPARENCY CARD

Session: {session\_id} | Avatar: {avatar\_id}

═══════════════════════════════════════════════════════════════════

AVDA Score: {avda\_pct['avda\_score']}% [{avda.classification.value}]

Accuracy: {avda\_pct['accuracy']}%

Confidence Interval: {avda\_pct['confidence\_interval'][0]}% - {avda\_pct['confidence\_interval'][1]}%

Source Coverage: {avda\_pct['source\_coverage']}%

Drift Risk: {avda\_pct['drift\_risk']}%

GROUND TRUTH SOURCES: {len(avatar\_dna.sources)} total

{chr(10).join(f"- {s.description} ({s.tier})" for s in avatar\_dna.sources[:5])}

LIMITATIONS:

{chr(10).join(f"⚠️ {lim}" for lim in avda.limitations) if avda.limitations else "✅ No significant limitations"}

═══════════════════════════════════════════════════════════════════

"""

return card

# ═══════════════════════════════════════════════════════════════════════════════

# CONVENIENCE FUNCTIONS

# ═══════════════════════════════════════════════════════════════════════════════

def create\_engine(config: Optional[Dict] = None) -> TactikEngine5\_3:

"""Factory function para crear engine"""

return TactikEngine5\_3(config)

async def quick\_simulation(engine: TactikEngine5\_3, profile: str, avatars: List[str],

queries: List[str], contexts: Optional[List[Dict]] = None) -> Dict[str, Any]:

"""Ejecuta simulación rápida completa"""

session = engine.create\_session(profile, avatars)

contexts = contexts or [{}] \* len(queries)

responses = []

for query, context in zip(queries, contexts):

response = await engine.process\_turn(session.session\_id, query, context)

responses.append(response)

tactik\_report = engine.generate\_tactik\_advisor\_report(session.session\_id)

transparency\_cards = {

avatar\_id: engine.generate\_transparency\_card(session.session\_id, avatar\_id)

for avatar\_id in avatars

}

return {

"session\_id": session.session\_id,

"responses": responses,

"tactik\_advisor\_report": tactik\_report,

"transparency\_cards": transparency\_cards

}

if \_\_name\_\_ == "\_\_main\_\_":

print("TACTIK 5.3 Premium Edition - Engine Loaded Successfully")

print("Import: from tactik\_engine\_5\_3 import create\_engine, TactikEngine5\_3")