IACADEMY

ENTERPRISE AGENT ENGINEERING

MÓDULO 03

Orquestación de Agentes

Principal

iacedemy.com — 2026

MÓDULO 03

Orquestación de Agentes

Nivel: Principal / Arquitecto

Autor: Alicia Sanz

Publicación: Julio 2026

Plataforma: iacedemy.com

Este material es parte del track Enterprise Agent Engineering de IAcademy.
Uso personal e intransferible.

> “El valor no está en un agente que funciona. Está en un sistema de agentes que se coordinan.”

2.1 Roles de Agentes en un Sistema Enterprise

Los 8 Roles

```
SUPERVISOR   — Coordina el flujo, decide qué agente actúa
PLANNER      — Descompone tareas complejas en sub-tareas
WORKER       — Ejecuta una tarea específica con tools
REVIEWER     — Evalúa la calidad del output de un worker
ROUTER       — Clasifica y enruta al agente correcto
MEMORY MGR   — Gestiona contexto compartido (scratchpad, RAG)
TOOL MGR     — Registra, valida y ejecuta tools
EVALUATOR    — Mide métricas y detecta degradación
```

Interface Base

```
from abc import ABC, abstractmethod
from dataclasses import dataclass, field
from typing import Any

@dataclass
class AgentMessage:
    from_role: str
    to_role: str
    content: Any
    metadata: dict = field(default_factory=dict)

class BaseAgent(ABC):
    def __init__(self, role: str, model: str, tools: list = None):
        self.role = role
        self.model = model
        self.tools = tools or []

    @abstractmethod
    async def process(self, message: AgentMessage) -> AgentMessage:
        """Procesa un mensaje y devuelve respuesta."""
        pass

    @abstractmethod
    async def can_handle(self, message: AgentMessage) -> bool:
        """Indica si este agente puede procesar el mensaje."""
        pass
```

Supervisor Pattern

```
class Supervisor(BaseAgent):
    def __init__(self, workers: dict[str, BaseAgent]):
        super().__init__(role="supervisor", model="claude-sonnet-4-6")
        self.workers = workers

    async def process(self, message: AgentMessage) -> AgentMessage:
        # 1. Analizar la tarea
        plan = await self._create_plan(message)

        # 2. Ejecutar plan paso a paso
        results = {}
        for step in plan.steps:
            worker = self.workers[step.worker_role]

            # Instrucciones ESPECÍFICAS al worker
            task = AgentMessage(
                from_role="supervisor",
                to_role=step.worker_role,
                content=step.instruction,  # No "handle this"
                metadata={"context": results}
            )

            result = await worker.process(task)

            # 3. Supervisor LEE el resultado
            quality = await self._evaluate_result(result, step)
            if not quality.acceptable:
                result = await self._retry_or_escalate(worker, task,
quality)

            results[step.id] = result

        # 4. Supervisor SINTETIZA respuesta final
        return await self._synthesize(results, message)
```

2.2 Grafos de Ejecución

LangGraph Avanzado: Pipeline SOC

```python
from langgraph.graph import StateGraph, END
from langgraph.checkpoint.sqlite import SqliteSaver
from typing import TypedDict, Annotated
import operator

class SOCState(TypedDict):
    alert: dict
    severity: str | None
    investigation: dict | None
    cti_results: list | None
    evidence: list | None
    ticket_id: str | None
    dashboard_updated: bool
    messages: Annotated[list, operator.add]

async def detect(state: SOCState) -> dict:
    """Clasificar severidad de la alerta."""
    alert = state["alert"]
    # LLM clasifica
    severity = await classify_alert(alert)
    return {"severity": severity, "messages": [f"Alert classified: {sever
ity}"]}

async def investigate(state: SOCState) -> dict:
    """Investigar la alerta: buscar IOCs, correlacionar."""
    alert = state["alert"]
    investigation = await run_investigation(alert)
    return {"investigation": investigation, "messages": [f"Investigation
complete: {len(investigation['iocs'])} IOCs found"]}

async def query_cti(state: SOCState) -> dict:
    """Consultar fuentes CTI para enriquecer IOCs."""
    iocs = state["investigation"]["iocs"]
    results = await check_threat_intel(iocs)
    return {"cti_results": results, "messages": [f"CTI enrichment: {len(r
esults)} matches"]}

async def generate_evidence(state: SOCState) -> dict:
    """Generar evidencias para compliance."""
    evidence = await create_evidence_package(state)
    return {"evidence": evidence, "messages": [f"Evidence package: {len(e
vidence)} items"]}

async def create_ticket(state: SOCState) -> dict:
    """Crear ticket en el sistema de gestión."""
    ticket_id = await create_jira_ticket(state)
    return {"ticket_id": ticket_id, "messages": [f"Ticket created: {ticke
t_id}"]}
```

```python
async def update_dashboard(state: SOCState) -> dict:
    """Actualizar dashboard SOC."""
    await push_to_dashboard(state)
    return {"dashboard_updated": True, "messages": ["Dashboard updated"]}

def should_investigate(state: SOCState) -> str:
    if state["severity"] in ("high", "critical"):
        return "investigate"
    return "log_and_close"

# Construir grafo
graph = StateGraph(SOCState)
graph.add_node("detect", detect)
graph.add_node("investigate", investigate)
graph.add_node("query_cti", query_cti)
graph.add_node("generate_evidence", generate_evidence)
graph.add_node("create_ticket", create_ticket)
graph.add_node("update_dashboard", update_dashboard)
graph.add_node("log_and_close", lambda s: {"messages": ["Low severity,
logged and closed"]})

graph.set_entry_point("detect")
graph.add_conditional_edges("detect", should_investigate, {
    "investigate": "investigate",
    "log_and_close": "log_and_close",
})
graph.add_edge("investigate", "query_cti")
graph.add_edge("query_cti", "generate_evidence")
graph.add_edge("generate_evidence", "create_ticket")
graph.add_edge("create_ticket", "update_dashboard")
graph.add_edge("update_dashboard", END)
graph.add_edge("log_and_close", END)

# Con checkpointing para recovery
checkpointer = SqliteSaver.from_conn_string("soc_checkpoints.db")
soc_pipeline = graph.compile(checkpointer=checkpointer)
```

State Machines vs Grafos

```
STATE MACHINE:
  - Estados finitos y predefinidos
  - Transiciones explícitas
  - Más predecible, menos flexible
  - Ideal: workflows regulatorios, aprobaciones

GRAFO (LangGraph):
  - Nodos como funciones
  - Edges condicionales
  - Checkpointing y recovery
  - Ideal: pipelines complejos, multi-step reasoning

DAG:
  - Grafo sin ciclos
  - Ejecución topológica
  - Paralelismo natural
  - Ideal: ETL, procesamiento batch
```

2.3 Arquitecturas de Orquestación

Sequential

```python
async def sequential_pipeline(input_data, agents: list):
    current = input_data
    for agent in agents:
        current = await agent.process(current)
    return current
```

Parallel (Fan-Out / Fan-In)

```
import asyncio

async def parallel_pipeline(input_data, agents: list, merger):
    # Fan-out
    tasks = [agent.process(input_data) for agent in agents]
    results = await asyncio.gather(*tasks, return_exceptions=True)

    # Filter errors
    valid = [r for r in results if not isinstance(r, Exception)]

    # Fan-in
    return await merger.process(valid)
```

Blackboard

```
class Blackboard:
    """Memoria compartida donde agentes leen y escriben."""

    def __init__(self):
        self._data = {}
        self._subscribers = {}

    async def write(self, key: str, value: Any, author: str):
        self._data[key] = {"value": value, "author": author, "ts":
time.time()}
        # Notificar suscriptores
        for callback in self._subscribers.get(key, []):
            await callback(key, value)

    async def read(self, key: str) -> Any:
        return self._data.get(key, {}).get("value")

    def subscribe(self, key: str, callback):
        self._subscribers.setdefault(key, []).append(callback)
```

Event-Driven

```
import redis.asyncio as redis

class EventBus:
    def __init__(self, redis_url: str):
        self.redis = redis.from_url(redis_url)

    async def publish(self, channel: str, event: dict):
        await self.redis.xadd(channel, event)

    async def subscribe(self, channel: str, group: str, consumer: str):
        # Crear grupo si no existe
        try:
            await self.redis.xgroup_create(channel, group, id="0",
mkstream=True)
        except redis.ResponseError:
            pass

        while True:
            messages = await self.redis.xreadgroup(
                groupname=group, consumername=consumer,
                streams={channel: ">"}, count=1, block=5000
            )
            for stream, msgs in messages:
                for msg_id, data in msgs:
                    yield data
                    await self.redis.xack(channel, group, msg_id)
```

2.4 Event Bus en Producción

Redis Streams para Agentes

```
# Productor: agente detector publica alertas
async def alert_producer(bus: EventBus):
    alert = {"type": "malware", "source_ip": "192.168.1.100", "severity":
 "high"}
    await bus.publish("alerts", alert)

# Consumidor: agente investigador procesa alertas
async def investigation_consumer(bus: EventBus):
    async for event in bus.subscribe("alerts", "investigators", "inv-1"):
        result = await investigate(event)
        await bus.publish("investigations", result)

# Consumidor: agente CTI enriquece investigaciones
async def cti_consumer(bus: EventBus):
    async for event in bus.subscribe("investigations", "cti-team", "cti-1
"):
        enriched = await enrich_with_cti(event)
        await bus.publish("enriched-alerts", enriched)
```

Temporal para Workflows Durables

```
from temporalio import workflow, activity
from datetime import timedelta

@activity.defn
async def classify_alert_activity(alert: dict) -> str:
    return await llm_classify(alert)

@activity.defn
async def investigate_activity(alert: dict) -> dict:
    return await run_siem_queries(alert)

@workflow.defn
class SOCWorkflow:
    @workflow.run
    async def run(self, alert: dict) -> dict:
        # Temporal garantiza at-least-once execution
        severity = await workflow.execute_activity(
            classify_alert_activity, alert,
            start_to_close_timeout=timedelta(seconds=30),
        )

        if severity in ("high", "critical"):
            investigation = await workflow.execute_activity(
                investigate_activity, alert,
                start_to_close_timeout=timedelta(minutes=5),
                retry_policy=RetryPolicy(maximum_attempts=3),
            )
            return {"severity": severity, "investigation": investigation}

        return {"severity": severity, "action": "logged"}
```

2.5 Caso Real: Pipeline SOC Completo

```
FLUJO:
  1. Alerta llega del SIEM
  2. Agente Detector clasifica severidad
  3. Si HIGH/CRITICAL:
     a. Agente Investigador analiza logs, correlaciona
     b. Agente CTI consulta feeds de amenazas
     c. Agente de Evidencias genera paquete compliance
     d. Agente de Tickets crea/actualiza en Jira
     e. Agente de Dashboard actualiza métricas
  4. Si LOW/MEDIUM:
     a. Log y auto-close
  5. Todo con trazas OpenTelemetry y budget tracking
```

Métricas del Pipeline

```
METRICS = {
    "alerts_processed_total": Counter,
    "alert_classification_duration": Histogram,
    "investigation_duration": Histogram,
    "false_positive_rate": Gauge,
    "mttr_seconds": Histogram,  # Mean Time To Respond
    "cost_per_alert_eur": Histogram,
    "escalation_rate": Gauge,
}
```

2.6 Ejercicios

Ejercicio 1: Supervisor Pattern

Implementa un Supervisor con 3 Workers (classifier, searcher, responder). El Supervisor debe: dar instrucciones específicas, leer resultados, y rechazar si calidad insuficiente.

Ejercicio 2: LangGraph Pipeline

Construye un pipeline con LangGraph de 5 nodos con conditional edges y checkpointing. Tema libre (SOC, soporte, content generation).

Ejercicio 3: Event Bus

Implementa un EventBus con Redis Streams donde 3 agentes se comunican: productor publica, 2 consumidores procesan en paralelo, resultados se consolidan.

Ejercicio 4: Blackboard Pattern

Implementa un Blackboard donde 4 agentes escriben resultados parciales y un Supervisor lee y sintetiza cuando todos han terminado.
