IACADEMY

AGENTIC ENGINEER & SDD

MÓDULO 05

Orquestación Multi-Agente

Avanzado

iacedemy.com — 2026

MÓDULO 05

Orquestación Multi-Agente

Nivel: Avanzado

Autor: Alicia Sanz

Publicación: Julio 2026

Plataforma: iacedemy.com

> “El 80% de los problemas se resuelven con 1 agente. El 20% restante justifica la complejidad de multi-agent.”

5.1 Cuándo Multi-Agent es Necesario (y Cuándo No)

Test de Necesidad

Antes de disenar un sistema multi-agente, responde estas preguntas:

```
1. Tu agente unico tiene > 10 tools?
   NO → Probablemente no necesitas multi-agent.
   SI → Considerar separar por dominio.

2. Las tools tienen niveles de privilegio diferentes?
   NO → Single agent con tool-level permissions.
   SI → Agentes separados con permisos distintos.

3. Hay tareas que pueden ejecutarse en paralelo?
   NO → Single agent con steps secuenciales.
   SI → Multi-agent con fan-out/fan-in.

4. Diferentes tareas necesitan diferentes modelos LLM?
   NO → Single agent, un modelo.
   SI → Multi-agent, modelo optimizado por worker.

5. Necesitas auditoria separada por componente?
   NO → Single agent con logging granular.
   SI → Multi-agent (trazas independientes).
```

El Coste de Multi-Agent

```
OVERHEAD POR CADA AGENTE ADICIONAL:

Latencia:    +2-5s por handoff (LLM call extra)
Tokens:      +500-2000 tokens por system prompt adicional
Complejidad: +1 punto de fallo, +1 config, +1 deploy
Debug:       Trazas distribuidas, más dificil reproducir bugs
Coste:       ~1.5-3x vs single agent para la misma tarea

REGLA: Si un single agent resuelve el 85%+ de los casos,
NO anadir agentes para el 15% restante.
Mejor: single agent + escalacion a humano para ese 15%.
```

5.2 Topologias Multi-Agente

Topologia 1: Pipeline (Secuencial)

```
Agent A → Agent B → Agent C → Output

Ejemplo: Content pipeline
  Researcher → Writer → Reviewer → Output

Caracteristicas:
  - Cada agente recibe el output del anterior
  - Simple de implementar y debuggear
  - Latencia = suma de todos los agentes
  - Fallo en cualquier paso = pipeline falla
```

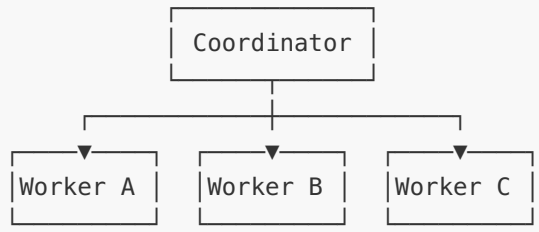
```python
class PipelineOrchestrator:
    def __init__(self, agents: list):
        self.agents = agents

    async def run(self, initial_input: str) -> str:
        current = initial_input
        for agent in self.agents:
            current = await agent.run(current)
        return current

# Uso
pipeline = PipelineOrchestrator([
    ResearchAgent(model="qwen3.5-27b"),
    WriterAgent(model="claude-sonnet-4-6"),
    ReviewerAgent(model="claude-haiku-4-5"),
])
result = await pipeline.run("Escribe un articulo sobre agentic
engineering")
```

Topologia 2: Fan-Out / Fan-In (Paralelo)

```
                ┌→ Agent A ─┐
Input ──────────┼→ Agent B ─┼──────→ Merger → Output
                └→ Agent C ─┘

Ejemplo: Code review
  Input (PR diff) → Lint Agent + Security Agent + Style Agent → Merge
findings

Caracteristicas:
  - Agentes trabajan en paralelo (rapido)
  - Merger sintetiza resultados
  - Latencia = max(agents) + merger
  - Un agente puede fallar sin bloquear los demas
```

```python
import asyncio

class FanOutOrchestrator:
    def __init__(self, agents: list, merger):
        self.agents = agents
        self.merger = merger

    async def run(self, input_data: str) -> str:
        # Fan-out: ejecutar todos en paralelo
        results = await asyncio.gather(*[
            agent.run(input_data) for agent in self.agents
        ], return_exceptions=True)

        # Filtrar errores
        valid_results = []
        for agent, result in zip(self.agents, results):
            if isinstance(result, Exception):
                valid_results.append(f"[{agent.name}] ERROR: {result}")
            else:
                valid_results.append(f"[{agent.name}] {result}")

        # Fan-in: merger sintetiza
        return await self.merger.run("\n\n".join(valid_results))

# Uso
orchestrator = FanOutOrchestrator(
    agents=[
        LintAgent(model="haiku"),
        SecurityAgent(model="sonnet"),
        StyleAgent(model="haiku"),
    ],
    merger=ReviewMergerAgent(model="sonnet"),
)
```

Topologia 3: Coordinator + Workers (Jerarquico)




```
Ejemplo: Soporte escalado
  Coordinator analiza query →
    Router Worker (clasificar) →
    Search Worker (buscar KB) →
    Response Worker (generar respuesta)

Caracteristicas:
  - Coordinator DECIDE, workers EJECUTAN
  - Coordinator ve resultados y redirige
  - Mas flexible que pipeline o fan-out
  - Coordinator es el punto critico (si falla, todo falla)
```

```python
class CoordinatorOrchestrator:
    def __init__(self, coordinator_model: str, workers: dict):
        self.coordinator = LLM(model=coordinator_model)
        self.workers = workers  # {name: WorkerAgent}

    async def run(self, user_input: str) -> str:
        messages = [
            {"role": "system", "content": self._coordinator_prompt()},
            {"role": "user", "content": user_input},
        ]

        for _ in range(10):  # Max delegations
            response = await self.coordinator.complete(
                messages=messages,
                tools=self._delegation_tools()
            )

            if not response.tool_calls:
                return response.text  # Respuesta final

            for tc in response.tool_calls:
                if tc.name == "delegate_to_worker":
                    worker_name = tc.arguments["worker"]
                    task = tc.arguments["task"]

                    worker = self.workers[worker_name]
                    result = await worker.run(task)

                    messages.append({
                        "role": "tool",
                        "tool_call_id": tc.id,
                        "content": result
                    })

        return "Limite de delegaciones alcanzado."

    def _coordinator_prompt(self) -> str:
        worker_list = "\n".join(
            f"- {name}: {w.description}"
            for name, w in self.workers.items()
        )
        return
f"""Eres un coordinador. Decides que worker usar para cada tarea.
Workers disponibles:
{worker_list}

REGLAS:
- LEE los resultados de cada worker antes de continuar.
- Da instrucciones ESPECIFICAS a cada worker, no vagas.
```

```
- Sintetiza resultados tu mismo, no delegues la sintesis.
- Si un worker falla, intenta otro approach o responde con lo que
tienes."""

    def _delegation_tools(self) -> list[dict]:
        return [{
            "type": "function",
            "function": {
                "name": "delegate_to_worker",
                "description": "Delegar una tarea especifica a un
worker",
                "parameters": {
                    "type": "object",
                    "properties": {
                        "worker": {
                            "type": "string",
                            "enum": list(self.workers.keys()),
                            "description": "Nombre del worker"
                        },
                        "task": {
                            "type": "string",
                            "description": "Instruccion especifica para
el worker"
                        }
                    },
                    "required": ["worker", "task"]
                }
            }
        }]
```

Topologia 4: Swarm (Descentralizado)

```
  Agent A ←→ Agent B
    ↕           ↕
  Agent C ←→ Agent D

Ejemplo: Debate/brainstorming
  Cada agente tiene un punto de vista diferente.
  Se pasan mensajes hasta llegar a consenso.

Caracteristicas:
  - Sin coordinador central
  - Agentes se comunican peer-to-peer
  - Dificil de controlar y predecir
  - Riesgo de loops infinitos (OBLIGATORIO max_rounds)
  - Pocas aplicaciones practicas en produccion
```

Decision: Que Topologia

```
Pipeline:    Cuándo los pasos son fijos y secuenciales.
             Content generation, ETL, data processing.

Fan-Out:     Cuándo multiples perspectivas independientes.
             Code review, analisis multi-criterio, búsqueda paralela.

Coordinator: Cuándo la decisión de que hacer depende del contexto.
             Soporte, triage, investigacion.

Swarm:       Cuándo necesitas debate/consenso (raro en produccion).
             Brainstorming, red teaming.

DEFAULT:     Coordinator + Workers. Es el más flexible
             y cubre el 90% de los casos multi-agent.
```

5.3 Comunicacion entre Agentes

Patron 1: Message Passing (Paso de Mensajes)

```
@dataclass
class AgentMessage:
    from_agent: str
    to_agent: str
    content: str
    metadata: dict = field(default_factory=dict)
    timestamp: float = field(default_factory=time.time)

class MessageBus:
    def __init__(self):
        self._queues: dict[str, asyncio.Queue] = {}

    def register(self, agent_name: str):
        self._queues[agent_name] = asyncio.Queue()

    async def send(self, message: AgentMessage):
        queue = self._queues.get(message.to_agent)
        if queue:
            await queue.put(message)

    async def receive(self, agent_name: str, timeout: float = 30) ->
AgentMessage:
        queue = self._queues[agent_name]
        return await asyncio.wait_for(queue.get(), timeout=timeout)
```

Cuándo usar: Agentes que necesitan comunicarse de forma asicrona. Topologias swarm o coordinator complejos.

Patron 2: Scratchpad Compartido

```python
class SharedScratchpad:
    """
    Memoria compartida entre agentes.
    Cada agente puede leer todo, pero solo escribe en su seccion.
    """

    def __init__(self):
        self._store: dict[str, dict[str, str]] = {}
        self._lock = asyncio.Lock()

    async def write(self, agent_name: str, key: str, value: str):
        async with self._lock:
            if agent_name not in self._store:
                self._store[agent_name] = {}
            self._store[agent_name][key] = value

    async def read(self, agent_name: str | None = None) -> dict:
        if agent_name:
            return self._store.get(agent_name, {})
        return dict(self._store)  # Todo el scratchpad

    async def summary(self) -> str:
        """Resumen para inyectar en contexto del coordinator."""
        lines = []
        for agent, data in self._store.items():
            for key, value in data.items():
                lines.append(f"[{agent}] {key}: {value[:100]}")
        return "\n".join(lines)
```

Cuándo usar: Coordinator + Workers. El coordinator ve todo, los workers escriben resultados.

Patron 3: Handoff (Transferencia)

```
async def handoff(
    from_agent: str,
    to_agent: str,
    context: dict,
    reason: str
) -> dict:
    """
    Transferir control de un agente a otro.
    El agente receptor recibe el contexto completo.
    """
    handoff_payload = {
        "from": from_agent,
        "reason": reason,
        "context": context,
        "conversation_history": context.get("messages", []),
        "scratchpad": context.get("scratchpad", {}),
        "timestamp": time.time(),
    }

    # El agente receptor arranca con el contexto del anterior
    target = agents[to_agent]
    result = await target.run_with_context(handoff_payload)

    return result
```

Cuándo usar: Cuándo un agente detecta que la tarea requiere otro especialista. Tipico en soporte: L1 → L2 → L3.

5.4 LangGraph: Implementacion Practica

Agente Multi-Step con LangGraph

```
"""
Ejemplo: Sistema de triage de tickets con LangGraph.
Coordinator clasifica → Worker busca → Worker responde o escala.
"""

from langgraph.graph import StateGraph, END
from langgraph.prebuilt import ToolNode
from typing import TypedDict, Annotated
import operator

# Estado compartido del grafo
class TicketState(TypedDict):
    messages: Annotated[list, operator.add]
    ticket_category: str | None
    search_results: list | None
    resolution: str | None
    escalated: bool

# Nodos del grafo
async def classify_ticket(state: TicketState) -> dict:
    """Clasificar el ticket en una categoria."""
    last_message = state["messages"][-1]["content"]

    response = await llm.complete(
        messages=[{
            "role": "system",
            "content": "Clasifica este ticket en: billing, technical,
account, feature_request. Responde SOLO con la categoria."
        }, {
            "role": "user",
            "content": last_message
        }]
    )

    return {"ticket_category": response.text.strip().lower()}

async def search_knowledge(state: TicketState) -> dict:
    """Buscar en KB basandose en la clasificacion."""
    query = state["messages"][-1]["content"]
    results = await search_kb(query, category=state["ticket_category"])
    return {"search_results": results}

async def generate_response(state: TicketState) -> dict:
    """Generar respuesta usando resultados de búsqueda."""
    results = state["search_results"]

    if not results or results[0]["score"] < 0.7:
        return {
            "resolution": None,
```

```
            "escalated": True,
            "messages": [{"role": "assistant", "content": "Escalando a
soporte humano..."}]
        }

    response = await llm.complete(
        messages=[{
            "role": "system",
            "content": f"Responde al usuario basandote en esta
informacion:\n{results[0]['content']}"
        }, {
            "role": "user",
            "content": state["messages"][-1]["content"]
        }]
    )

    return {
        "resolution": response.text,
        "escalated": False,
        "messages": [{"role": "assistant", "content": response.text}]
    }

def should_escalate(state: TicketState) -> str:
    """Decision: escalar o responder."""
    if state.get("escalated"):
        return "escalate"
    return "respond"

# Construir el grafo
graph = StateGraph(TicketState)

graph.add_node("classify", classify_ticket)
graph.add_node("search", search_knowledge)
graph.add_node("respond", generate_response)
graph.add_node("escalate", lambda s: {"messages": [{"role": "system", "co
ntent": "Ticket escalado a L2"}]})

graph.set_entry_point("classify")
graph.add_edge("classify", "search")
graph.add_edge("search", "respond")
graph.add_conditional_edges("respond", should_escalate, {
    "escalate": "escalate",
    "respond": END,
})
graph.add_edge("escalate", END)

# Compilar y ejecutar
app = graph.compile()
result = await app.ainvoke({
    "messages": [{"role": "user", "content": "No puedo hacer login desde
```

```
ayer"}],
    "ticket_category": None,
    "search_results": None,
    "resolution": None,
    "escalated": False,
})
```

Checkpointing (Recuperacion de Fallos)

```
from langgraph.checkpoint.sqlite import SqliteSaver

# Persistir estado para recovery
checkpointer = SqliteSaver.from_conn_string("checkpoints.db")
app = graph.compile(checkpointer=checkpointer)

# Cada ejecucion se guarda automaticamente
# Si el proceso crashea, se puede resumir desde el ultimo checkpoint
config = {"configurable": {"thread_id": "ticket-123"}}
result = await app.ainvoke(initial_state, config)
```

5.5 Aislamiento entre Agentes

Niveles de Aislamiento

```
NIVEL 1: IN-PROCESS (sin aislamiento)
  - Todos los agentes en el mismo proceso Python
  - Comparten memoria, pueden interferir
  - Rapido pero peligroso con codigo no confiable
  Usar: Workers internos simples, scope confiable

NIVEL 2: ASYNC TASKS (aislamiento logico)
  - asyncio tasks separadas
  - Comparten event loop pero no estado
  - Comunicacion via queues
  Usar: La mayoria de casos multi-agent

NIVEL 3: SUBPROCESS (aislamiento de proceso)
  - Cada agente en su propio proceso
  - Comunicacion via pipes/sockets
  - Si uno crashea, no tumba a los demas
  Usar: Agentes con dependencias conflictivas

NIVEL 4: CONTAINER (aislamiento completo)
  - Cada agente en su propio container Docker
  - Filesystem, red, y procesos separados
  - Comunicacion via HTTP/gRPC
  Usar: Agentes que ejecutan codigo no confiable (sandbox)

NIVEL 5: WORKTREE (para code agents)
  - Cada agente trabaja en una copia del repositorio
  - Git worktree: mismo repo, branch separado
  - Sin conflictos de filesystem
  Usar: Agentes que modifican archivos de codigo
```

Patron: Worker Pool con Aislamiento

```python
import asyncio
from dataclasses import dataclass

@dataclass
class WorkerConfig:
    name: str
    model: str
    tools: list[str]
    max_concurrent: int = 3
    timeout: int = 60

class WorkerPool:
    def __init__(self, configs: list[WorkerConfig]):
        self.workers = {}
        for config in configs:
            self.workers[config.name] = {
                "config": config,
                "semaphore": asyncio.Semaphore(config.max_concurrent),
            }

    async def execute(self, worker_name: str, task: str) -> str:
        worker = self.workers[worker_name]
        config = worker["config"]

        async with worker["semaphore"]:  # Limitar concurrencia
            try:
                agent = create_agent(
                    model=config.model,
                    tools=config.tools,
                )
                result = await asyncio.wait_for(
                    agent.run(task),
                    timeout=config.timeout
                )
                return result
            except asyncio.TimeoutError:
                return f"Worker {worker_name} timeout tras {config.timeou
t}s"

# Configuracion
pool = WorkerPool([
    WorkerConfig(name="classifier", model="haiku", tools=[],
max_concurrent=10),
    WorkerConfig(name="searcher", model="qwen3.5", tools=["search_kb"],
max_concurrent=5),
    WorkerConfig(name="responder", model="sonnet", tools=["search_kb", "e
scalate"], max_concurrent=3),
])
```

5.6 Anti-Patterns Multi-Agent

Anti-Pattern 1: Lazy Coordinator

```
# MAL: coordinator delega sin leer ni especificar
coordinator_prompt = """
Tienes workers disponibles. Delega la tarea al worker apropiado.
Basandote en sus findings, responde al usuario.
"""
# "Basandote en sus findings" = el coordinator NO lee los resultados.
# Genera respuesta sin verificar calidad/relevancia.

# BIEN: coordinator que LEE y ESPECIFICA
coordinator_prompt = """
Workers: classifier, searcher, responder.

PROCESO:
1. Primero, delega a classifier con instruccion:
   "Clasifica en: billing/technical/account/feature. Solo la categoria."
2. LEE el resultado del classifier.
3. Delega a searcher con instruccion ESPECIFICA:
   "Busca en la categoria X: [query exacta derivada del usuario]"
4. LEE los resultados del searcher.
5. Si los resultados son relevantes (score > 0.7), genera TU la
respuesta.
6. Si no son relevantes, escala a humano.
NUNCA delegues la generacion de respuesta final a un worker.
"""
```

Anti-Pattern 2: Over-Decomposition

```
MAL: 7 agentes para un FAQ bot
  InputParser → IntentClassifier → EntityExtractor
  → KBSearcher → AnswerGenerator → ToneChecker → OutputFormatter

BIEN: 1 agente con 2 tools
  FAQAgent(tools=[search_kb, escalate])

El primer diseño tiene 7x latencia, 7x coste, 7x puntos de fallo.
Para el mismo resultado.
```

Anti-Pattern 3: Shared Mutable State

```python
# MAL: agentes modifican el mismo diccionario sin control
shared_state = {"findings": [], "score": 0}

# Worker A escribe findings
# Worker B lee findings mientras A todavia esta escribiendo
# Race condition → datos corruptos

# BIEN: cada worker escribe en su seccion, coordinator lee
scratchpad = SharedScratchpad()
# Worker A: scratchpad.write("worker_a", "findings", data_a)
# Worker B: scratchpad.write("worker_b", "findings", data_b)
# Coordinator: scratchpad.read()  # Lee todo, sin conflictos
```

5.7 Modelo de Costes Multi-Agent

Calcular Coste

```python
def estimate_multi_agent_cost(
    topology: str,
    agents: list[dict],
    queries_per_day: int
) -> dict:
    """
    Estimar coste mensual de un sistema multi-agent.
    """
    total_tokens_per_query = 0

    for agent in agents:
        # System prompt + herencia de contexto
        overhead_tokens = agent["system_prompt_tokens"] + agent.get("cont
ext_tokens", 0)
        # Tokens por ejecucion (input + output)
        execution_tokens = agent["avg_input_tokens"] + agent["avg_output_
tokens"]

        total_tokens_per_query += overhead_tokens + execution_tokens

    # Overhead de coordinacion (si aplica)
    if topology in ("coordinator", "fan-out"):
        coordination_overhead = 1.3  # 30% extra por mensajes de
coordinacion
    else:
        coordination_overhead = 1.0

    total_tokens_per_query *= coordination_overhead

    monthly_tokens = total_tokens_per_query * queries_per_day * 30
    monthly_cost = monthly_tokens * avg_cost_per_token(agents)

    return {
        "tokens_per_query": int(total_tokens_per_query),
        "monthly_tokens": int(monthly_tokens),
        "monthly_cost_eur": round(monthly_cost, 2),
        "cost_per_query_eur": round(monthly_cost / (queries_per_day * 30)
, 4),
    }

# Ejemplo
cost = estimate_multi_agent_cost(
    topology="coordinator",
    agents=[
        {"name": "coordinator", "system_prompt_tokens": 500, "avg_input_t
okens": 300, "avg_output_tokens": 200},
        {"name": "classifier", "system_prompt_tokens": 200, "avg_input_to
kens": 200, "avg_output_tokens": 50},
        {"name": "searcher", "system_prompt_tokens": 300, "avg_input_toke
```

```
ns": 400, "avg_output_tokens": 300},
    ],
    queries_per_day=500
)
# Resultado: ~4.5M tokens/mes, ~€13.50/mes con Haiku
```

Optimizacion: Model Routing

```
No todos los agentes necesitan el mismo modelo.

Coordinator:  Modelo capaz (Sonnet/Qwen3.5-27B)
              Necesita razonar, decidir, sintetizar.

Classifier:   Modelo rapido y barato (Haiku/Phi-4)
              Tarea simple, pocas categorias.

Searcher:     Sin LLM (solo embedding + vector search)
              No necesita generacion de texto.

Responder:    Modelo medio (Sonnet/Qwen3.5-27B)
              Necesita generar texto de calidad.

Resultado: 60-70% reduccion de coste vs usar Sonnet para todo.
```

5.8 Ejercicios Practicos

Ejercicio 1: Topologia

Para cada caso, elegir topologia y justificar: 1. Sistema que genera reportes de ventas semanales (datos de 3 fuentes) 2. Agente de code review (lint + security + tests + style) 3. Chatbot de soporte multi-idioma (ES, EN, PT, FR) 4. Sistema de screening de CVs (extraer datos + scoring + matching)

Ejercicio 2: LangGraph

Implementar con LangGraph un agente que: 1. Recibe pregunta del usuario 2. Clasifica en: facil (responder directo) o compleja (investigar) 3. Si facil: responder con LLM 4. Si

compleja: buscar en KB → si no encuentra, buscar en web → responder 5. Review final: verificar que la respuesta es relevante

Ejercicio 3: Coordinator Prompt

Escribir el system prompt para un coordinator que: - Tiene 3 workers: researcher, analyst, writer - Debe producir un informe de mercado - Debe LEE y VERIFICAR cada resultado antes de continuar - No puede delegar más de 5 veces

Ejercicio 4: Coste

Calcular coste mensual para: - 1000 queries/dia - Topologia: coordinator + 3 workers - Coordinator: Sonnet (input: 300 tokens, output: 200) - Workers: Haiku (input: 200 cada uno, output: 150 cada uno) - Comparar con: single agent Sonnet (input: 800, output: 500)
