IACADEMY

AGENTIC ENGINEER & SDD

MÓDULO 03

Diseño de Agentes

Avanzado

iacedemy.com — 2026

MÓDULO 03

Diseño de Agentes

Nivel: Avanzado

Autor: Alicia Sanz

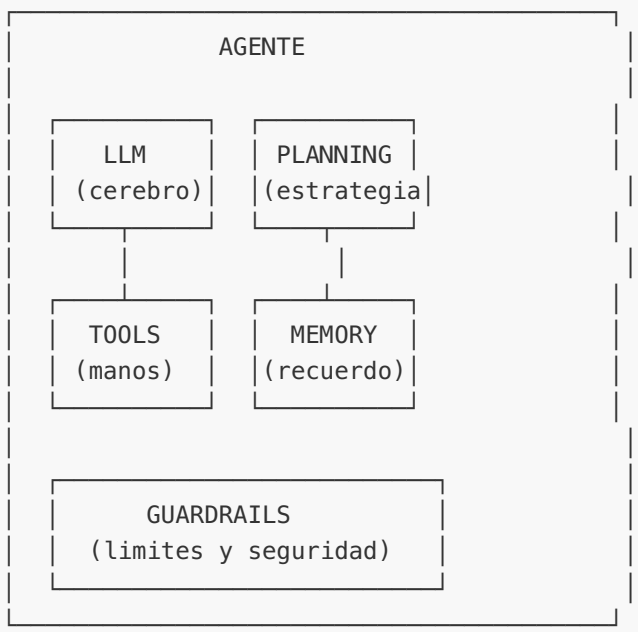
Publicación: Julio 2026

Plataforma: iacedemy.com

> “Un agente bien disenado es aburrido. Hace exactamente lo que debe, nada mas, nada menos.”

3.1 Anatomia de un Agente

Los 5 Componentes



Componente	Responsabilidad	Ejemplo
LLM	Razonamiento, generacion de texto	Qwen3.5-27B, Claude Sonnet
Planning	Decidir que hacer y en que orden	ReAct, Plan-and-Execute
Tools	Interactuar con el mundo exterior	APIs, bases de datos, búsqueda
Memory	Retener informacion entre turnos	Context window, RAG, scratchpad
Guardrails	Limitar comportamiento	HITL, circuit breakers, budgets

Chatbot vs Copilot vs Agente

```
CHATBOT:
  - Input → LLM → Output
  - Sin tools, sin planning, sin memory persistente
  - Ejemplo: FAQ bot con respuestas predefinidas

COPILOT:
  - Input → LLM + Tools → Output (humano decide)
  - Tools disponibles pero humano controla el flujo
  - Ejemplo: GitHub Copilot, Claude en IDE

AGENTE:
  - Input → Planning → [Tool calls en loop] → Output
  - El agente decide que tools usar, en que orden, cuantas veces
  - Loop autonomo hasta completar tarea o alcanzar limite
  - Ejemplo: Devin, agente SOC que triagea alertas
```

Decision: Cuándo NO Necesitas un Agente

```
Usa un PROMPT SIMPLE si:
  - La tarea es input → output sin tools
  - No hay decisión condicional
  - Un solo LLM call basta

Usa un CHAIN si:
  - Pasos fijos y predeterminados (siempre A → B → C)
  - No hay branching condicional
  - Ejemplo: extraer texto → resumir → traducir

Usa un AGENTE si:
  - Los pasos dependen del resultado anterior
  - Hay branching (si X, hacer Y; si no, hacer Z)
  - Se necesitan multiples tool calls con logica
  - El numero de pasos no es predecible

Usa MULTI-AGENTE si:
  - Un solo agente necesitaria >10 tools
  - Hay dominios de conocimiento separados
  - Se necesita paralelismo real
  - Requisito de separacion de privilegios
```

3.2 Diseño de Tools

Tool = API Contract

Cada tool es un contrato entre el agente y el mundo exterior.

```python
# Anatomia de una tool bien disenada

from dataclasses import dataclass
from enum import Enum

class RiskLevel(Enum):
    LOW = "low"        # Solo lectura, sin side effects
    MEDIUM = "medium"  # Escribe datos, reversible
    HIGH = "high"      # Side effects irreversibles (enviar email,
cobrar)
    CRITICAL = "critical"  # Destructivo (borrar datos, revocar acceso)

@dataclass
class ToolMetadata:
    name: str                    # verbo_objeto (search_knowledge,
create_ticket)
    description: str             # 1-2 frases. QUE hace, no COMO.
    risk_level: RiskLevel
    timeout_seconds: int         # Siempre. Sin timeout = sin circuit
breaker.
    is_concurrency_safe: bool    # Puede ejecutarse en paralelo?
    requires_hitl: bool          # Necesita aprobacion humana?
    idempotent: bool             # Llamar 2x produce el mismo resultado?
    cost_per_call: float | None  # Coste estimado (para budget tracking)
```

Reglas de Diseño de Tools

Regla 1: Una tool, una responsabilidad

```python
# MAL: tool que hace demasiado
def manage_customer_tool(action, customer_id, data=None):
    """Crear, leer, actualizar, o borrar cliente."""
    if action == "create": ...
    elif action == "read": ...
    elif action == "update": ...
    elif action == "delete": ...

# BIEN: tools separadas con risk_level apropiado
def get_customer_tool(customer_id: str) -> Customer:
    """Obtener datos de un cliente por ID."""
    # risk_level: LOW

def update_customer_tool(customer_id: str, fields: dict) -> Customer:
    """Actualizar campos de un cliente existente."""
    # risk_level: MEDIUM

def delete_customer_tool(customer_id: str) -> bool:
    """Eliminar permanentemente un cliente y sus datos."""
    # risk_level: CRITICAL, requires_hitl: True
```

Regla 2: Descripciones para el LLM, no para humanos

```python
# MAL: descripcion tecnica
"""Ejecuta query SQL parametrizada contra PostgreSQL via connection
pool."""

# BIEN: descripcion orientada a uso
"""Buscar tickets de soporte de un cliente.
Usar cuando el usuario pregunte por el estado de su problema
o quiera ver tickets anteriores. Devuelve los ultimos N tickets
ordenados por fecha, más reciente primero."""
```

Regla 3: Parametros minimos, outputs ricos

```python
# MAL: muchos parámetros opcionales confunden al LLM
def search_tool(
    query: str,
    filters: dict | None = None,
    sort_by: str = "relevance",
    sort_order: str = "desc",
    page: int = 1,
    page_size: int = 10,
    include_metadata: bool = True,
    highlight: bool = False,
    fuzzy: bool = True,
    min_score: float = 0.0,
):
    ...

# BIEN: parámetros esenciales, defaults inteligentes
def search_knowledge_base(query: str, top_k: int = 5) -> list[SearchResul
t]:
    """Buscar en la base de conocimiento por query semantica.
    Devuelve los top_k resultados más relevantes con score y fuente."""
    ...

@dataclass
class SearchResult:
    content: str        # Texto del resultado
    source: str         # URL o referencia de la fuente
    score: float        # 0.0 a 1.0, relevancia semantica
    section: str        # Seccion del documento
```

Regla 4: Errores claros, no excepciones opacas

```python
# MAL: excepcion generica
def get_customer(customer_id: str):
    customer = db.query(Customer, customer_id)
    if not customer:
        raise Exception("Not found")  # El LLM no sabe que hacer con esto
    return customer

# BIEN: resultado tipado con caso de error
@dataclass
class ToolResult:
    success: bool
    data: dict | None = None
    error: str | None = None
    suggestion: str | None = None  # Que deberia hacer el agente si falla

def get_customer(customer_id: str) -> ToolResult:
    customer = db.query(Customer, customer_id)
    if not customer:
        return ToolResult(
            success=False,
            error=f"Cliente con ID {customer_id} no encontrado",
            suggestion="Verificar el ID con el usuario o buscar por
email"
        )
    return ToolResult(success=True, data=customer.to_dict())
```

Regla 5: Idempotencia siempre que sea posible

```python
# MAL: no idempotente, duplica si se llama 2 veces
def create_ticket(title: str, description: str) -> Ticket:
    return db.insert(Ticket(title=title, description=description))

# BIEN: idempotente con dedup key
def create_ticket(
    title: str,
    description: str,
    idempotency_key: str
) -> Ticket:
    existing = db.query(Ticket, idempotency_key=idempotency_key)
    if existing:
        return existing  # Ya creado, devolver existente
    return db.insert(Ticket(
        title=title,
        description=description,
        idempotency_key=idempotency_key
    ))
```

Cuántas Tools? Regla del 7 (mas/menos 2)

```
1-3 tools:   Single agent, scope acotado. Optimo.
4-7 tools:   Single agent, scope medio. Manejable.
8-12 tools:  Considerar separar en 2 agentes o agrupar tools.
13+ tools:   Multi-agent obligatorio. Un agente no puede razonar
             bien sobre 13+ opciones en cada turno.

Excepcion: si las tools son mutuamente excluyentes
(solo se usa 1 por turno), el limite es mayor.
```

3.3 Patrones de Planning

Patron 1: ReAct (Reasoning + Acting)

```
Thought: Necesito buscar informacion sobre el error del usuario
Action: search_knowledge_base("error 503 al hacer login")
Observation: [resultados de la búsqueda]
Thought: El resultado sugiere que es un problema de rate limiting
Action: get_ticket_history(customer_id="cust_123")
Observation: [3 tickets previos, todos sobre rate limiting]
Thought: Es un problema recurrente, debo escalar con contexto
Action: escalate_to_human(reason="rate_limiting_recurrente", ...)
```

Cuándo usar: Tareas donde el agente necesita razonar paso a paso y cada paso depende del anterior. Mayoria de casos.

Pros: Simple, debuggeable, funciona bien con la mayoria de LLMs. **Contras:** Puede ser lento si necesita muchos pasos. Riesgo de loops.

Patron 2: Plan-and-Execute

```
Plan:
  1. Buscar en KB el error reportado
  2. Verificar si hay tickets previos del mismo cliente
  3. Si es recurrente, escalar. Si no, responder con solucion.

Execute:
  Step 1: search_knowledge_base("error 503") → [resultados]
  Step 2: get_ticket_history("cust_123") → [3 tickets]
  Step 3: Plan dice escalar → escalate_to_human(...)
```

Cuándo usar: Tareas complejas donde vale la pena planificar antes de actuar. El plan puede revisarse/aprobarse antes de ejecutar.

Pros: Mas predecible. El plan se puede mostrar al usuario antes. **Contras:** Overhead del paso de planning. El plan puede no sobrevivir al contacto con la realidad.

Patron 3: Reflection

```
Attempt 1:
  Action: responder con solucion generica
  Reflection: La respuesta no cita fuentes y es demasiado vaga.
              El usuario podria no confiar en ella.

Attempt 2:
  Action: buscar en KB, responder citando fuente especifica
  Reflection: Mejor. Pero no he verificado si el usuario tiene
              tickets previos sobre esto.

Attempt 3:
  Action: buscar KB + verificar historial + responder con contexto
  Reflection: Completo. Tiene fuente, contexto, y es accionable.
  → Output final
```

Cuándo usar: Tareas donde la calidad del output es critica y vale la pena "pensar dos veces". Code review, analisis de seguridad, redaccion importante.

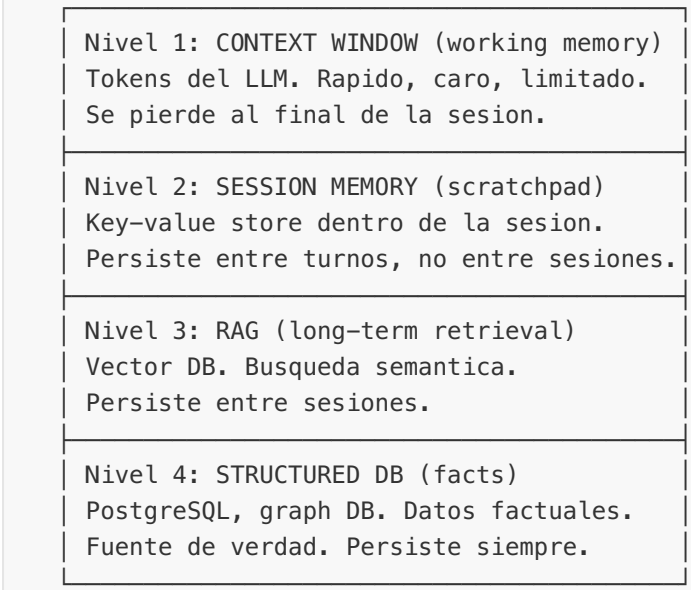
Pros: Output de mayor calidad. Auto-correccion. **Contras:** 2-3x tokens. Solo vale la pena si el coste del error es alto.

Decision Tree: Que Patron Usar

```
La tarea es simple (1-2 tool calls)?
  → ReAct

La tarea es compleja y el plan es mostrable al usuario?
  → Plan-and-Execute

La calidad del output es critica y el coste del error es alto?
  → Reflection

Necesitas combinar?
  → Plan-and-Execute + Reflection en el paso final
```

3.4 Arquitectura de Memoria

4 Niveles de Memoria



Decision Tree: Que Nivel de Memoria Usar

```
El dato se necesita SOLO en este turno?
  → Context window (ya esta ahi)

El dato se necesita EN ESTA SESION pero no en futuras?
  → Scratchpad / session state
  Ejemplo: resumen parcial, resultados intermedios,
           preferencias del usuario en esta conversacion

El dato se necesita EN FUTURAS SESIONES y es texto no estructurado?
  → RAG (vector DB)
  Ejemplo: documentos, conversaciones pasadas, notas

El dato se necesita EN FUTURAS SESIONES y es estructurado/factual?
  → Base de datos relacional
  Ejemplo: perfil del usuario, historial de compras, configuracion
```

Patron: Scratchpad para Agentes

```python
class AgentScratchpad:
    """
    Memoria de trabajo del agente dentro de una sesion.
    Mas eficiente que meter todo en el context window.
    """

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

    def write(self, key: str, value: str) -> str:
        """Guardar un dato para uso posterior en esta sesion."""
        self._store[key] = value
        return f"Guardado: {key}"

    def read(self, key: str) -> str:
        """Leer un dato guardado previamente."""
        return self._store.get(key, f"No encontrado: {key}")

    def summarize(self) -> str:
        """Resumen de todo lo guardado (para inyectar en contexto)."""
        if not self._store:
            return "Scratchpad vacio."
        lines = [f"- {k}: {v[:100]}" for k, v in self._store.items()]
        return "Scratchpad actual:\n" + "\n".join(lines)
```

Patron: Context Compression

```
def compress_context(messages: list[dict], max_tokens: int) -> list[dict]
:
    """
    Cuándo el contexto supera el limite, comprimir mensajes antiguos.

    Estrategia:
    1. System prompt: NUNCA comprimir
    2. Ultimos 3 turnos: NUNCA comprimir (working memory)
    3. Tool results > 500 tokens: resumir a 100 tokens
    4. Mensajes antiguos: consolidar en resumen
    """
    system = [m for m in messages if m["role"] == "system"]
    recent = messages[-6:]  # Ultimos 3 turnos (user + assistant)
    old = messages[len(system):-6]

    # Comprimir mensajes antiguos en un resumen
    if old:
        summary = llm.summarize(old, max_tokens=200)
        compressed = [{"role": "system", "content": f"Resumen previo: {su
mmary}"}]
    else:
        compressed = []

    return system + compressed + recent
```

3.5 HITL: Human-in-the-Loop

Cuándo Requirir HITL

```
SIEMPRE HITL (no negociable):
  - Operaciones destructivas (borrar datos, revocar acceso)
  - Comunicaciones externas (enviar email, publicar)
  - Transacciones financieras (cobrar, reembolsar)
  - Cambios de permisos (elevar privilegios)
  - Deploy a produccion

HITL CONDICIONAL (segun confidence/risk):
  - Clasificacion con confidence < threshold
  - Primera vez que el agente ejecuta un tipo de accion
  - Datos sensibles involucrados

NUNCA HITL (auto-approve):
  - Lecturas de datos
  - Busquedas en KB
  - Calculos internos
  - Logging y metricas
```

Patron: Dual-Track Permissions

```
from enum import Enum

class PermissionTrack(Enum):
    AUTO = "auto"       # Se ejecuta sin aprobacion
    SUPERVISED = "supervised"  # Requiere aprobacion humana

class Permission:
    def __init__(self, tool_name: str, risk_level: RiskLevel):
        self.tool_name = tool_name
        self.risk_level = risk_level

    @property
    def track(self) -> PermissionTrack:
        if self.risk_level in (RiskLevel.LOW,):
            return PermissionTrack.AUTO
        if self.risk_level in (RiskLevel.MEDIUM,):
            # Auto si hay historial de ejecuciones exitosas
            return PermissionTrack.AUTO if self._has_trust() else
PermissionTrack.SUPERVISED
        return PermissionTrack.SUPERVISED  # HIGH, CRITICAL siempre
supervisado

    def _has_trust(self) -> bool:
        """El agente ha ejecutado esta tool 10+ veces sin incidentes."""
        history = get_execution_history(self.tool_name)
        recent = [h for h in history if h.age_days < 30]
        return len(recent) >= 10 and all(h.success for h in recent)
```

Patron: HITL con Timeout y Fallback

```
import asyncio

async def execute_with_hitl(
    tool_name: str,
    params: dict,
    timeout_minutes: int = 30,
    fallback: str = "skip"  # "skip" | "escalate" | "default_value"
) -> ToolResult:
    """
    Solicitar aprobacion humana con timeout.
    Si no hay respuesta en timeout, ejecutar fallback.
    """
    approval_request = await send_approval_request(
        tool=tool_name,
        params=params,
        channel="telegram",  # Notificar por Telegram
    )

    try:
        approval = await asyncio.wait_for(
            wait_for_approval(approval_request.id),
            timeout=timeout_minutes * 60
        )

        if approval.approved:
            return await execute_tool(tool_name, params)
        else:
            return ToolResult(
                success=False,
                error=f"Rechazado por {approval.user}: {approval.reason}"
            )

    except asyncio.TimeoutError:
        if fallback == "skip":
            return ToolResult(success=False, error="HITL timeout, accion
omitida")
        elif fallback == "escalate":
            return await escalate_to_manager(tool_name, params)
        elif fallback == "default_value":
            return ToolResult(success=True, data=get_safe_default(tool_na
me))
```

3.6 Circuit Breakers

Por que Son Obligatorios

Sin circuit breakers, un agente puede: - Entrar en loop infinito (gastando tokens sin limite) - Reintentar una API caida indefinidamente - Acumular errores en cascada

```python
from dataclasses import dataclass, field
from time import time

@dataclass
class CircuitBreaker:
    """
    3 estados: CLOSED (normal), OPEN (bloqueado), HALF_OPEN (probando).
    """
    max_failures: int = 5
    reset_timeout_seconds: int = 60

    _failure_count: int = field(default=0, init=False)
    _last_failure_time: float = field(default=0, init=False)
    _state: str = field(default="CLOSED", init=False)

    def can_execute(self) -> bool:
        if self._state == "CLOSED":
            return True

        if self._state == "OPEN":
            # Verificar si paso suficiente tiempo para probar de nuevo
            if time() - self._last_failure_time > self.reset_timeout_seco
nds:
                self._state = "HALF_OPEN"
                return True
            return False

        if self._state == "HALF_OPEN":
            return True  # Permitir un intento de prueba

        return False

    def record_success(self):
        self._failure_count = 0
        self._state = "CLOSED"

    def record_failure(self):
        self._failure_count += 1
        self._last_failure_time = time()
        if self._failure_count >= self.max_failures:
            self._state = "OPEN"

    @property
    def state(self) -> str:
        return self._state


# Uso en el agente
class Agent:
```

```python
    def __init__(self):
        self.circuit_breakers = {
            "search_kb": CircuitBreaker(max_failures=3,
reset_timeout_seconds=30),
            "external_api": CircuitBreaker(max_failures=2,
reset_timeout_seconds=120),
        }
        self.max_iterations = 10  # NUNCA loop infinito
        self.token_budget = 50_000  # Limite de tokens por ejecucion

    async def run(self, query: str):
        iterations = 0
        tokens_used = 0

        while iterations < self.max_iterations:
            iterations += 1

            # Verificar budget
            if tokens_used > self.token_budget:
                return self.fallback_response("Token budget agotado")

            # Seleccionar accion
            action = self.plan_next_action(query)

            if action.type == "final_answer":
                return action.answer

            # Verificar circuit breaker
            cb = self.circuit_breakers.get(action.tool)
            if cb and not cb.can_execute():
                # Tool bloqueada, usar fallback
                return self.fallback_response(
                    f"Tool {action.tool} temporalmente no disponible"
                )

            # Ejecutar tool
            try:
                result = await self.execute_tool(action.tool,
action.params)
                if cb:
                    cb.record_success()
                tokens_used += result.tokens
            except Exception as e:
                if cb:
                    cb.record_failure()
                continue

        return self.fallback_response("Limite de iteraciones alcanzado")
```

Budget por Ejecucion

```python
@dataclass
class ExecutionBudget:
    max_tokens: int = 50_000
    max_tool_calls: int = 15
    max_duration_seconds: int = 120
    max_cost_eur: float = 0.50  # Para APIs de pago

    _tokens_used: int = 0
    _tool_calls: int = 0
    _start_time: float = field(default_factory=time)
    _cost_eur: float = 0.0

    def check(self) -> tuple[bool, str]:
        if self._tokens_used >= self.max_tokens:
            return False, "Token budget agotado"
        if self._tool_calls >= self.max_tool_calls:
            return False, "Limite de tool calls alcanzado"
        if time() - self._start_time >= self.max_duration_seconds:
            return False, "Timeout de ejecucion"
        if self._cost_eur >= self.max_cost_eur:
            return False, "Budget de coste agotado"
        return True, "OK"
```

3.7 Single Agent vs Multi-Agent

Decision Framework

```
Pregunta 1: Cuántas tools necesita?
  ≤ 7  → Single agent probablemente basta
  > 7  → Considerar multi-agent

Pregunta 2: Hay dominios de conocimiento separados?
  No  → Single agent
  Si  → Un agente por dominio

Pregunta 3: Se necesitan privilegios diferentes?
  No  → Single agent
  Si  → Agentes separados con permisos distintos

Pregunta 4: Hay paralelismo real (no secuencial)?
  No  → Single agent con steps
  Si  → Multi-agent con coordinador

Pregunta 5: Hay requisito de auditoria por componente?
  No  → Single agent más simple
  Si  → Multi-agent (trazabilidad separada)
```

Anti-Pattern: Multi-Agent Prematuro

```
SITUACION: Bot de soporte que busca en KB y responde.

MAL (over-engineering):
  Coordinator Agent → Router Agent → Search Agent → Answer Agent → QA
Agent
  5 agentes, 5x latencia, 5x tokens, 5x complejidad

BIEN:
  1 agente con 3 tools (search_kb, get_history, escalate)
  Simple, rapido, debuggeable

REGLA: si tu single agent funciona con <= 7 tools y <= 10
iteraciones por query, NO necesitas multi-agent.
```

Patron Multi-Agent Correcto: Coordinator + Workers

```python
class Coordinator:
    """
    El coordinador LEE resultados de workers y DECIDE.
    NUNCA delega sin especificar exactamente QUE hacer.
    """

    def __init__(self, workers: dict[str, Worker]):
        self.workers = workers

    async def handle(self, task: Task) -> Result:
        # 1. Analizar la tarea (el coordinador PIENSA)
        analysis = await self.analyze(task)

        # 2. Asignar a workers con instrucciones ESPECIFICAS
        # MAL: "Based on your findings, handle this"
        # BIEN: instrucciones exactas

        if analysis.needs_search:
            search_result = await self.workers["search"].execute(
                query=analysis.search_query,
                filters=analysis.search_filters,
                max_results=5
            )

            # 3. El coordinador LEE el resultado (no delega la lectura)
            relevant = self._filter_relevant(search_result, task)

        if analysis.needs_classification:
            classification = await self.workers["classifier"].execute(
                text=task.input,
                categories=["billing", "technical", "account", "other"]
            )

        # 4. El coordinador SINTETIZA y decide
        response = await self.synthesize(
            task=task,
            search_results=relevant,
            classification=classification
        )

        return response

class Worker:
    """
    Worker con scope acotado. Solo sabe hacer UNA cosa.
    """
    def __init__(self, name: str, tools: list, system_prompt: str):
        self.name = name
        self.tools = tools  # Max 3-5 tools por worker
```

```
        self.system_prompt = system_prompt
        self.scratchpad = AgentScratchpad()  # Aislado
```

Aislamiento entre Workers

```
IN-PROCESS (mismo proceso Python):
  - Rapido (no overhead de red)
  - Compartir memoria (cuidado con race conditions)
  - Para workers ligeros con scope muy acotado

SUBPROCESS (proceso separado):
  - Aislamiento de memoria
  - Si un worker crashea, no tumba al coordinador
  - Para workers con dependencias diferentes

WORKTREE (repo separado, para code agents):
  - Aislamiento total del filesystem
  - Cada worker tiene su copia del repo
  - Para agentes que modifican archivos

CONTAINER (Docker):
  - Aislamiento completo (filesystem, red, procesos)
  - Para workers que ejecutan codigo no confiable
  - Overhead más alto (~1-2s startup)
```

3.8 Tool Composition Patterns

Patron 1: Sequential

```
Tool A → resultado → Tool B → resultado → Tool C → output

Ejemplo:
  search_kb("error login")
    → resultados
  get_customer_history("cust_123")
    → historial
  create_response(resultados + historial)
    → respuesta final
```

Patron 2: Parallel (Fan-Out / Fan-In)

```
        ┌→ Tool A → resultado A ─┐
Input ──┼→ Tool B → resultado B ─┼→ Merge → Output
        └→ Tool C → resultado C ─┘

Ejemplo:
  Paralelo:
    search_kb("error login")        → resultados KB
    get_customer_history("cust_123") → historial
    check_system_status("login")     → estado sistema

  Merge:
    Combinar los 3 resultados en respuesta contextualizada
```

```python
import asyncio

async def parallel_tools(tasks: list[tuple[str, dict]]) -> list[ToolResul
t]:
    """Ejecutar multiples tools en paralelo."""
    results = await asyncio.gather(*[
        execute_tool(name, params) for name, params in tasks
    ])
    return results

# Uso
results = await parallel_tools([
    ("search_kb", {"query": "error login"}),
    ("get_customer_history", {"customer_id": "cust_123"}),
    ("check_system_status", {"service": "login"}),
])
```

Patron 3: Conditional (Branching)

```
Input → Tool A → resultado
                    ├─ si condicion X → Tool B → Output 1
                    └─ si condicion Y → Tool C → Output 2

Ejemplo:
  classify_query("quiero cancelar mi cuenta")
    → categoria: "account_cancellation"
    → risk: HIGH
    → requiere HITL: SI

  if risk == HIGH:
    escalate_to_human(reason="cancelacion")
  else:
    handle_automatically()
```

Patron 4: Retry with Backoff

```python
async def retry_tool(
    tool_name: str,
    params: dict,
    max_retries: int = 3,
    backoff_seconds: float = 1.0
) -> ToolResult:
    """Retry con exponential backoff. Circuit breaker externo."""

    for attempt in range(max_retries):
        try:
            result = await execute_tool(tool_name, params)
            if result.success:
                return result
        except Exception as e:
            if attempt == max_retries - 1:
                return ToolResult(
                    success=False,
                    error=f"Fallido tras {max_retries} intentos: {e}"
                )

        await asyncio.sleep(backoff_seconds * (2 ** attempt))

    return ToolResult(success=False, error="Max retries agotado")
```

3.9 Observabilidad del Agente

Que Instrumentar

```python
from opentelemetry import trace
from opentelemetry.trace import StatusCode

tracer = trace.get_tracer("agent-support-001")

class InstrumentedAgent:
    async def run(self, query: str):
        with tracer.start_as_current_span("agent.run") as span:
            span.set_attribute("agent.id", "AGENT-SUPPORT-001")
            span.set_attribute("agent.version", "1.0")
            span.set_attribute("query.length", len(query))

            iterations = 0
            total_tokens = 0

            while not done:
                iterations += 1

                with tracer.start_as_current_span("agent.iteration") as
iter_span:
                    iter_span.set_attribute("iteration.number",
iterations)

                    # LLM call
                    with tracer.start_as_current_span("llm.call") as
llm_span:
                        response = await self.llm.complete(messages)
                        llm_span.set_attribute("llm.model", self.model)
                        llm_span.set_attribute("llm.tokens.input",
response.input_tokens)
                        llm_span.set_attribute("llm.tokens.output",
response.output_tokens)
                        total_tokens += response.total_tokens

                    # Tool call
                    if response.tool_call:
                        with tracer.start_as_current_span("tool.call")
as tool_span:
                            tool_span.set_attribute("tool.name",
response.tool_call.name)
                            tool_span.set_attribute("tool.risk_level",
tool.risk_level)
                            result = await self.execute_tool(response.too
l_call)
                            tool_span.set_attribute("tool.success",
result.success)

            span.set_attribute("agent.iterations", iterations)
            span.set_attribute("agent.total_tokens", total_tokens)
```

```
            span.set_attribute("agent.resolution", "auto" if not
escalated else "escalated")
```

Metricas Clave

```
POR AGENTE:
  - resolution_rate:     % tareas completadas sin escalacion
  - avg_iterations:      media de iteraciones por tarea
  - avg_latency:         tiempo total de ejecucion
  - avg_tokens:          tokens consumidos por ejecucion
  - avg_cost:            coste por ejecucion
  - error_rate:          % ejecuciones con error
  - circuit_breaker_trips: veces que se abrio un circuit breaker

POR TOOL:
  - call_count:          veces invocada
  - success_rate:        % llamadas exitosas
  - avg_latency:         tiempo de respuesta
  - error_types:         distribucion de errores

POR SESION:
  - user_satisfaction:   CSAT, rating
  - turns_to_resolve:    turnos hasta resolucion
  - escalation_reason:   por que se escalo
```

3.10 Ejercicios Practicos

Ejercicio 1: Tool Audit

Dado este conjunto de tools, identificar problemas y redisenar:

```
tools = [
    "manage_user(action, user_id, data)",      # action: create/read/
update/delete
    "send_notification(type, target, message)", # type: email/sms/push
    "query_database(sql)",                      # SQL directo
    "do_everything(task_description)",           # "catch-all"
]
```

Ejercicio 2: Memory Architecture

Disenar la arquitectura de memoria para un agente de ventas que: - Recuerda conversaciones pasadas con cada cliente - Tiene acceso al catalogo de productos (5000 items) - Necesita context de pricing actualizado al minuto - Sesiones de 30+ minutos con multiples temas

Ejercicio 3: Circuit Breaker Config

Configurar circuit breakers para un agente que usa: - API interna (latencia 50ms, muy fiable) - API externa de terceros (latencia 2s, caida 1x/semana) - LLM self-hosted (latencia 3s, GPU shared) - Base de datos (latencia 10ms, critica)

Ejercicio 4: Single vs Multi-Agent

Para cada caso, decidir y justificar: 1. Bot de FAQ con 5 categorias de preguntas 2. Sistema de code review automatico (lint + security + style + tests) 3. Agente de onboarding que configura cuenta, envia emails, y programa call 4. Pipeline de analisis de CVs (extraer datos + scoring + recomendacion)
