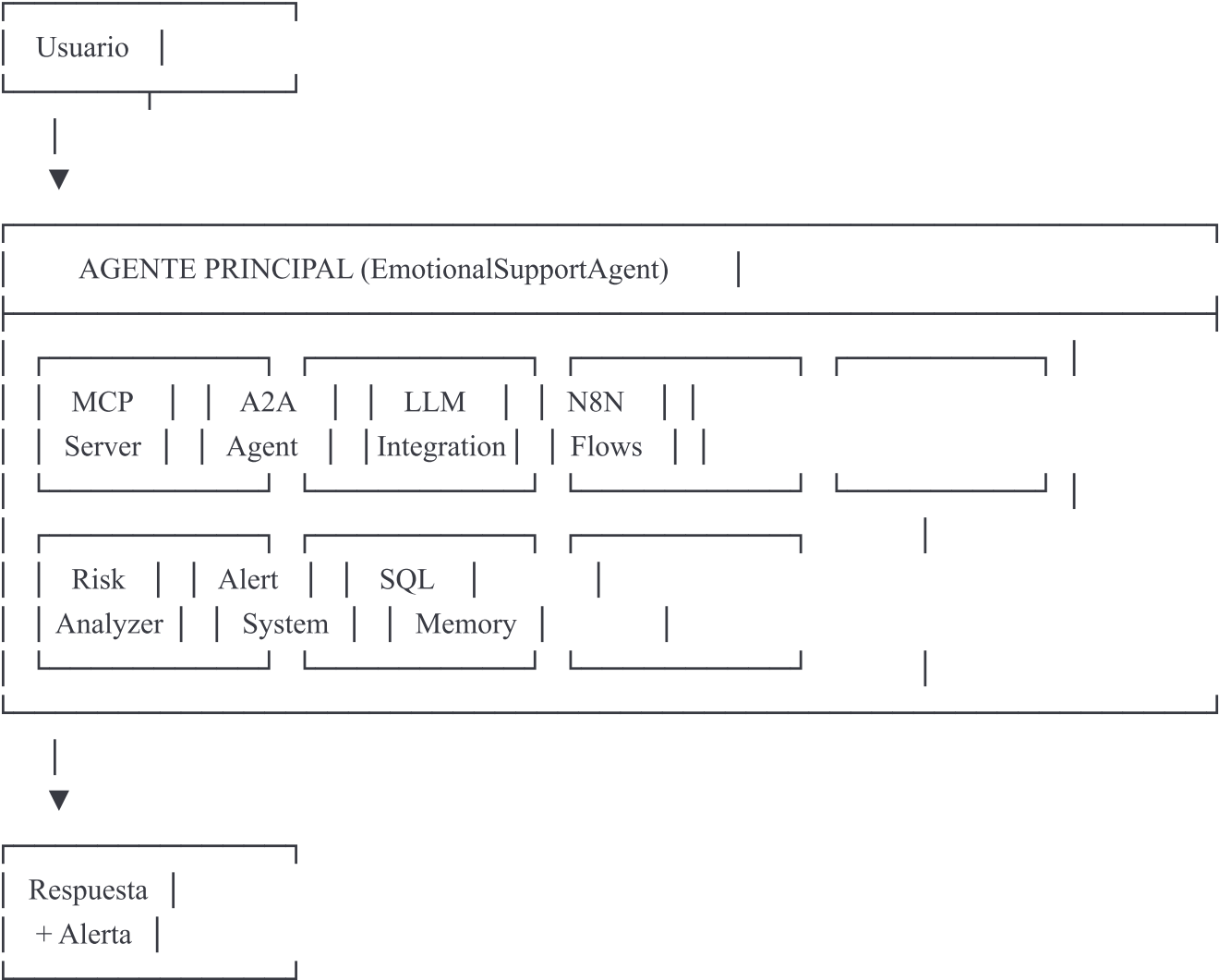


EXPLICACIÓN DETALLADA DEL CÓDIGO

Analizador de Riesgo Emocional - Día 2 (25%)

ARQUITECTURA GENERAL

El sistema está compuesto por **8 componentes principales** que trabajan de forma integrada:



1 MCP SERVER (Model Context Protocol)

Clase: MCPServer

Propósito: Expone las capacidades del agente como servicios que otros sistemas pueden consumir.

Métodos:

`__init__(self, port: int = 8080)`

Función: Constructor de la clase **Parámetros:**

- port: Puerto donde escucha el servidor (default: 8080)

Qué hace:

1. Guarda el puerto
2. Define un diccionario de capacidades disponibles:
 - analyze_risk: Analiza riesgo emocional de texto
 - generate_response: Genera respuesta empática
 - activate_alert: Activa protocolo de alerta
3. Imprime confirmación de inicialización

Ejemplo de uso:



python

```
mcp = MCPServer(port=8080)
# Output:  MCP Server inicializado (puerto 8080)
```

`get_capabilities(self)` -> Dict

Función: Retorna las capacidades del servidor **Retorna:** Diccionario con:

- protocol_version: Versión del protocolo MCP
- server_name: Nombre del servidor
- capabilities: Diccionario de capacidades disponibles

Qué hace:

- Empaqueta la información del servidor en formato estándar
- Permite que otros agentes descubran qué servicios ofrece

Ejemplo de uso:



python

```
caps = mcp.get_capabilities()
print(caps)

# {
#   "protocol_version": "1.0",
#   "server_name": "emotional-risk-analyzer",
#   "capabilities": {
#     "analyze_risk": {...},
#     "generate_response": {...},
#     "activate_alert": {...}
#   }
# }
```

`handle_request(self, method: str, params: Dict) -> Dict`

Función: Procesa solicitudes de otros sistemas **Parámetros:**

- `method`: Nombre del método a ejecutar
- `params`: Parámetros de la solicitud

Qué hace:

1. Valida que el método exista en las capacidades
2. Si no existe, retorna error
3. Si existe, retorna confirmación con timestamp

Retorna: Diccionario con resultado

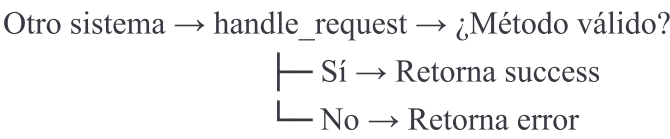
Ejemplo de uso:



python

```
response = mcp.handle_request("analyze_risk", {"text": "estoy triste"})
# {"status": "success", "method": "analyze_risk", "timestamp": "2025-10-21T..."}
```

Flujo:



2 A2A AGENT (Agent-to-Agent)

Clase: A2AAgent

Propósito: Permite comunicación bidireccional entre agentes usando el protocolo A2A y procesamiento con LLM (Ollama).

Métodos:

`__init__(self, model: str = "llama3.2:1b")`

Función: Constructor del agente A2A **Parámetros:**

- `model`: Modelo LLM a usar (default: llama3.2:1b)

Qué hace:

1. Guarda el modelo a usar
2. Genera un ID único de 8 caracteres para el agente
3. Intenta conectar con Ollama:
 - Si está disponible: marca `available = True`
 - Si no está disponible: usa modo simulado
4. Imprime estado de inicialización

Variables de instancia:

- `self.model`: Modelo LLM
- `self.agent_id`: ID único del agente
- `self.available`: Bool indicando si Ollama está disponible

Ejemplo:



python

```
agent = A2AAgent(model="llama3.2:1b")
# ✅ A2A Agent inicializado (Ollama: llama3.2:1b)
# 🇺🇸 Agent ID: a3f5c912
```

`send_to_agent(self, target_id: str, content: Dict) -> Dict`

Función: Envía mensaje a otro agente **Parámetros:**

- `target_id`: ID del agente destinatario
- `content`: Contenido del mensaje (diccionario)

Qué hace:

1. Crea un mensaje estructurado con:
 - `from`: ID de este agente
 - `to`: ID del destinatario
 - `timestamp`: Momento del envío
 - `content`: Contenido del mensaje
 - `protocol`: Versión del protocolo (A2A-v1)

2. Imprime confirmación del envío
3. Retorna el mensaje completo

Retorna: Diccionario con el mensaje A2A

Ejemplo:



python

```
message = agent.send_to_agent("coordinator", {
    "type": "risk_assessment",
    "user_id": "maria",
    "risk_level": "ALTO"
})
# 🚩 A2A: a3f5c912 → coordinator
# Retorna:
# {
#   "from": "a3f5c912",
#   "to": "coordinator",
#   "timestamp": "2025-10-21T10:30:00",
#   "content": {"type": "risk_assessment", ...},
#   "protocol": "A2A-v1"
# }
```

Flujo:



Agente local → send_to_agent → Crea mensaje estructurado →
Imprime confirmación → Retorna mensaje

receive_from_agent(self, message: Dict) -> Dict

Función: Recibe y procesa mensaje de otro agente **Parámetros:**

- message: Mensaje A2A recibido

Qué hace:

1. Imprime confirmación de recepción
2. Extrae el remitente del mensaje
3. Crea respuesta confirmando la recepción
4. Retorna la respuesta

Retorna: Diccionario con respuesta

Ejemplo:



python

```
incoming = {
    "from": "coordinator",
    "to": "a3f5c912",
    "content": {"command": "status"}}
}
response = agent.receive_from_agent(incoming)
# 🇺🇸 A2A: coordinator → a3f5c912
# Retorna:
# {
#   "from": "a3f5c912",
#   "to": "coordinator",
#   "status": "received"
# }
```

Flujo:



Mensaje entrante → receive_from_agent → Imprime confirmación →
Crea respuesta → Retorna confirmación

3 LLM INTEGRATION

📦 Clase: LLMIntegration

Propósito: Integra modelos de lenguaje (LLM) para generar respuestas empáticas. Usa Ollama localmente con soporte para API keys.

Métodos:

`__init__(self, api_key: str = None)`

Función: Constructor de la integración LLM **Parámetros:**

- `api_key`: API key opcional (por defecto busca en variable de entorno)

Qué hace:

1. Obtiene API key de parámetro o variable de entorno `OLLAMA_API_KEY`
2. Define modelo por defecto: `llama3.2:1b`
3. Intenta conectar con Ollama:
 - Si conecta: `available = True`
 - Si falla: usa respuestas predefinidas (fallback)

4. Imprime estado de disponibilidad


Variables de instancia:

- self.api_key: API key para autenticación
- self.model: Modelo LLM a usar
- self.available: Bool indicando disponibilidad

Ejemplo:



python

```
llm = LLMIntegration(api_key="mi-api-key")
#  LLM Integration: Ollama (llama3.2:1b)
```

```
generate_empathy_response(self, message: str, risk_level: str, name: str) -> str
```

Función: Genera respuesta empática personalizada usando LLM **Parámetros:**

- message: Mensaje original del usuario
- risk_level: Nivel de riesgo detectado (ALTO, MODERADO, BAJO, NEUTRAL)
- name: Nombre del usuario

Qué hace:

1. Verifica si LLM está disponible
2. Si NO está disponible → usa _fallback_response()
3. Si SÍ está disponible:
 - Construye prompt detallado con contexto
 - Envía prompt a Ollama
 - Procesa respuesta del LLM
 - Limpia espacios y retorna
4. Si hay error en LLM → usa fallback

Retorna: String con la respuesta empática

Prompt usado:



Eres un consejero de apoyo emocional. El usuario {name} escribió:
"{message}"

Nivel de riesgo detectado: {risk_level}

- Genera una respuesta empática en 2-3 oraciones que:
- Sea genuina y humana
 - Reconozca el dolor sin minimizarlo
 - Ofrezca apoyo apropiado al nivel de riesgo

Ejemplo:



python

```
response = llm.generate_empathy_response(  
    message="Me siento muy solo y triste",  
    risk_level TEXT,           -- Nivel detectado  
    indicators TEXT            -- Indicadores (JSON)  
)
```

Propósito: Almacena cada evaluación de riesgo realizada.

Tabla 2: alerts (Alertas activadas)



sql

```
CREATE TABLE IF NOT EXISTS alerts (  
    alert_id TEXT PRIMARY KEY,      -- ID único de alerta  
    user_id TEXT,                  -- ID del usuario  
    risk_level TEXT,               -- Nivel de riesgo  
    priority TEXT,                 -- Prioridad (CRÍTICA/ALTA)  
    timestamp TEXT                 -- Momento de activación  
)
```

Propósito: Registra todas las alertas activadas.

Tabla 3: flow_executions (Ejecuciones de workflows)



sql


```
CREATE TABLE IF NOT EXISTS flow_executions (  
    execution_id TEXT PRIMARY KEY,      -- ID único de ejecución  
    flow_name TEXT,                    -- Nombre del flujo  
    timestamp TEXT,                    -- Momento de ejecución  
    status TEXT                        -- Estado (completed/failed)  
)
```

Propósito: Registra cada vez que se ejecuta un workflow.

Flujo:



- _init_db() → Conectar a SQLite →
- Crear tabla assessments si no existe →
- Crear tabla alerts si no existe →
- Crear tabla flow_executions si no existe →
- Commit cambios → Cerrar conexión

```
save_assessment(self, user_id: str, message: str, analysis: Dict)
```

Función: Guarda una evaluación de riesgo en la base de datos **Parámetros:**

- user_id: ID del usuario evaluado
- message: Mensaje original del usuario
- analysis: Diccionario con el análisis de riesgo completo

Qué hace:

- Conecta a la base de datos
- Prepara query INSERT con los datos
- Convierte indicators de lista a JSON string
- Inserta el registro con timestamp actual
- Hace commit de la transacción
- Cierra la conexión

Query ejecutado:



sql

```
INSERT INTO assessments (user_id, timestamp, message, risk_level, indicators)  
VALUES (?, ?, ?, ?, ?)
```

Ejemplo:



python

```
memory.save_assessment(  
    user_id="maria_123",  
    message="Me siento muy triste",  
    analysis={  
        "risk_level": "MODERADO",  
        "indicators": ["Aislamiento", "Desesperanza"]  
    }  
)  
  
# Guarda en BD:  
# id=1, user_id="maria_123", timestamp="2025-10-21T...",  
# message="Me siento muy triste", risk_level="MODERADO",  
# indicators=['Aislamiento', 'Desesperanza']
```

save_alert(self, alert: Dict)

Función: Guarda una alerta activada en la base de datos **Parámetros:**

- alert: Diccionario con información completa de la alerta

Qué hace:

1. Conecta a la base de datos
2. Extrae los campos necesarios del diccionario alert
3. Inserta registro en tabla alerts
4. Hace commit
5. Cierra conexión

Query ejecutado:



sql

```
INSERT INTO alerts (alert_id, user_id, risk_level, priority, timestamp)  
VALUES (?, ?, ?, ?, ?)
```

Ejemplo:



python

```
alert = {  
    "alert_id": "a7f3c8d2",  
    "user_id": "maria_123",  
    "risk_level": "ALTO",  
    "priority": "CRÍTICA",  
    "timestamp": "2025-10-21T14:30:00"  
}  
  
memory.save_alert(alert)  
  
# Guarda en BD con todos los campos del diccionario
```

save_flow_execution(self, execution: Dict)

Función: Guarda registro de ejecución de workflow **Parámetros:**

- execution: Diccionario con información de la ejecución

Qué hace:

1. Conecta a la base de datos
2. Extrae campos del diccionario execution
3. Inserta registro en tabla flow_executions
4. Hace commit
5. Cierra conexión

Query ejecutado:



sql

```
INSERT INTO flow_executions (execution_id, flow_name, timestamp, status)  
VALUES (?, ?, ?, ?)
```

Ejemplo:



python

```
execution = {  
    "execution_id": "e7a2b4c1",  
    "flow_name": "Análisis Automático de Riesgo",  
    "timestamp": "2025-10-21T14:30:00",  
    "status": "completed"  
}  
  
memory.save_flow_execution(execution)
```

get_stats(self) -> Dict

Función: Obtiene estadísticas generales del sistema **Parámetros:** Ninguno

Qué hace: Ejecuta 4 consultas SQL para obtener métricas clave:

Consulta 1: Total de evaluaciones



sql

SELECT COUNT(*) FROM assessments

Consulta 2: Casos de alto riesgo



sql

SELECT COUNT(*) FROM assessments WHERE risk_level = 'ALTO'

Consulta 3: Total de alertas



sql

SELECT COUNT(*) FROM alerts

Consulta 4: Total de workflows ejecutados



sql

SELECT COUNT(*) FROM flow_executions

Retorna: Diccionario con todas las estadísticas

Ejemplo:



python

```
stats = memory.get_stats()
#{
#  "total_assessments": 145,
#  "high_risk_cases": 12,
#  "total_alerts": 23,
#  "flow_executions": 290
#}
```

Flujo:



get_stats() → Conectar BD →
 Contar assessments → Contar alto riesgo →
 Contar alertas → Contar flows →
 Construir diccionario → Cerrar conexión →
 Retornar estadísticas

8 EMOTIONAL SUPPORT AGENT (AGENTE PRINCIPAL)

 **Clase:** EmotionalSupportAgent

Propósito: Componente central que integra TODOS los sistemas y coordina el flujo completo de procesamiento.

Métodos:

__init__(self)

Función: Constructor del agente principal **Parámetros:** Ninguno

Qué hace:

- 1. Genera ID único de sesión (8 caracteres)
- 2. Inicializa TODOS los componentes obligatorios:



python

```
self.mcp = MCPServer()           # Servidor MCP
self.a2a = A2AAgent()            # Agente A2A
self.llm = LLMIntegration()      # Integración LLM
self.flows = N8NFlowManager()    # Workflows N8N
```

- 3. Inicializa componentes de análisis:



python

```
self.analyzer = RiskAnalyzer()    # Analizador de riesgo
self.alerts = AlertSystem()       # Sistema de alertas
self.memory = SQLMemory()         # Memoria SQL
```

4. Imprime banner de confirmación con ID de sesión

Variables de instancia:

- self.session_id: ID único de la sesión
- self.mcp: Instancia del servidor MCP
- self.a2a: Instancia del agente A2A
- self.llm: Instancia de integración LLM
- self.flows: Instancia del gestor de workflows
- self.analyzer: Instancia del analizador
- self.alerts: Instancia del sistema de alertas
- self.memory: Instancia de la memoria SQL

Ejemplo:



python

```
agent = EmotionalSupportAgent()
# ✅ MCP Server inicializado (puerto 8080)
# ✅ A2A Agent inicializado (Ollama: llama3.2:1b)
# 🗂️ Agent ID: a3f5c912
# ✅ LLM Integration: Ollama (llama3.2:1b)
# ✅ N8N Flow Manager inicializado
# • 3 flujos creados
#
# =====
# 💙 AGENTE LISTO - Sesión: b7e4d9a1
# =====
```

```
process_message(self, message: str, user_name: str = "Usuario") -> Dict
```

Función: Método principal que procesa un mensaje completo con TODOS los componentes **Parámetros:**

- message: Mensaje del usuario a analizar
- user_name: Nombre del usuario (default: "Usuario")

Qué hace - FLUJO COMPLETO EN 8 PASOS:

PASO 1: Preparación



python

```
user_id = user_name.lower().replace(" ", "_") # Normaliza nombre
print(f"📄 Procesando mensaje de {user_name}")
```

Convierte nombre a ID válido (ej: "María López" → "maría_lópez")

PASO 2: Ejecutar Workflow de Análisis



python

```
flow_exec = self.flows.execute_flow("auto_analysis", {
    "message": message,
    "user_id": user_id
})
self.memory.save_flow_execution(flow_exec)
```

- Activa el workflow "Análisis Automático de Riesgo"
- Pasa el mensaje como entrada
- Guarda la ejecución en BD

PASO 3: Analizar Riesgo



python

```
print(f"🔍 Analizando riesgo...")
analysis = self.analyzer.analyze(message)
print(f"{analysis['urgency']} Nivel: {analysis['risk_level']}")
```

- Usa el RiskAnalyzer para evaluar el mensaje
- Detecta nivel de riesgo e indicadores
- Muestra resultado en consola

PASO 4: Generar Respuesta Empática con LLM



python

```
print(f"💙 Generando respuesta empática...")
response = self.llm.generate_empathy_response(
    message, analysis['risk_level'], user_name
)
```

- Usa LLM para generar respuesta personalizada
- Toma en cuenta el nivel de riesgo
- Personaliza con el nombre del usuario

PASO 5: Ejecutar Workflow de Respuesta



python

```
flow_exec2 = self.flows.execute_flow("empathy_response", {
    "risk_level": analysis['risk_level'],
    "response": response
})
self.memory.save_flow_execution(flow_exec2)
```

- Activa workflow "Generación de Respuesta Empática"
- Valida y procesa la respuesta
- Guarda ejecución

PASO 6: Activar Alerta (si necesario)



python


```

alert = None

if analysis['requires_alert']:
    print(f"🚨 Activando protocolo de alerta...")
    alert = self.alerts.activate(
        user_id, analysis['risk_level'], analysis['indicators']
    )
    # Mostrar prioridad y acciones
    self.memory.save_alert(alert)

    # Ejecutar workflow de notificación
    flow_exec3 = self.flows.execute_flow("notification", {
        "alert_id": alert['alert_id'],
        "priority": alert['priority']
    })
    self.memory.save_flow_execution(flow_exec3)

```

- Si es ALTO o MODERADO, activa alerta
- Determina prioridad y acciones
- Ejecuta workflow de notificación
- Guarda todo en BD

PASO 7: Comunicación A2A



python

```

a2a_msg = self.a2a.send_to_agent("coordinator", {
    "type": "assessment",
    "user_id": user_id,
    "risk_level": analysis['risk_level']
})

```

- Notifica a agente coordinador
- Envía información del análisis
- Usa protocolo A2A

PASO 8: Exponer vía MCP



python

```
mcp_response = self.mcp.handle_request("analyze_risk", {
    "text": message,
    "user_id": user_id
})
```

- Registra la operación en el servidor MCP
- Permite que otros sistemas consulten

PASO 9: Guardar en Memoria SQL



python

```
self.memory.save_assessment(user_id, message, analysis)
if alert:
    self.memory.save_alert(alert)
```

- Guarda evaluación en BD
- Guarda alerta si se activó

PASO 10: Mostrar Respuesta



python

```
print(f"💖 RESPUESTA:")
print(response)
```

- Muestra la respuesta empática generada

Retorna: Diccionario completo con todos los resultados



python

```
return {
    "analysis": analysis,          # Análisis de riesgo
    "response": response,         # Respuesta empática
    "alert": alert,               # Alerta (o None)
    "a2a_message": a2a_msg,      # Mensaje A2A enviado
    "mcp_status": mcp_response['status'], # Estado MCP
    "flows_executed": 2 if not alert else 3 # Num workflows
}
```

Ejemplo completo:



python

```
result = agent.process_message(  
    "Me siento muy triste y solo, no sé qué hacer",  
    "María"  
)
```

```
# Output en consola:  
# =====  
# 📄 Procesando mensaje de María  
# =====  
#  
# ⚡ Flow ejecutado: Análisis Automático de Riesgo  
# 🔍 Analizando riesgo...  
#  
# ⚠️ ATENCIÓN Nivel: MODERADO  
# 📊 Indicadores: Aislamiento  
#  
# 💙 Generando respuesta empática...  
# ⚡ Flow ejecutado: Generación de Respuesta Empática  
#  
# 🚨 Activando protocolo de alerta...  
#   Prioridad: ALTA  
#   ✅ Consejero notificado  
#   ✅ Recursos compartidos  
#   ✅ Seguimiento en 24h  
# ⚡ Flow ejecutado: Notificación a Equipos  
# 📄 A2A: a3f5c912 → coordinator  
#  
# =====  
# 💙 RESPUESTA:  
# =====  
# Hola María, puedo ver que estás lidiando con algo pesado. Es válido  
# sentirse así. No tienes que enfrentarlo solo/a.  
# =====  
  
# result = {  
#   "analysis": {...},  
#   "response": "Hola María...",  
#   "alert": {"alert_id": "...", "priority": "ALTA", ...},  
#   "a2a_message": {...},  
#   "mcp_status": "success",  
#   "flows_executed": 3  
# }
```

Diagrama de flujo completo:



```
process_message() →
├─ Normalizar user_id
├─ Ejecutar workflow "auto_analysis"
├─ Analizar riesgo (RiskAnalyzer)
├─ Generar respuesta (LLM)
├─ Ejecutar workflow "empathy_response"
├─ ¿Requiere alerta?
│   ├─ SÍ → Activar alerta
│   │   └─ → Ejecutar workflow "notification"
│   │   └─ → Guardar alerta en BD
│   └─ NO → Continuar
├─ Enviar mensaje A2A
├─ Registrar en MCP
├─ Guardar assessment en BD
└─ Retornar resultado completo
```

show_status(self)

Función: Muestra estado completo del sistema con estadísticas **Parámetros:** Ninguno

Qué hace:

- 1. Obtiene estadísticas de la BD
- 2. Imprime tabla formateada con:
 - Estado de cada componente (MCP, A2A, LLM, N8N)
 - IDs y puertos
 - Disponibilidad de LLM
 - Número de workflows activos
 - Estadísticas de uso:
 - Total de evaluaciones
 - Casos de alto riesgo
 - Alertas activas
 - Workflows ejecutados

Ejemplo de output:





ESTADO DEL SISTEMA



Componentes:



MCP Server - Puerto 8080



A2A Agent - ID a3f5c912



LLM - Ollama



N8N Flows - 3 activos



Estadísticas:

Evaluaciones: 145

Alto riesgo: 12

Alertas: 23

Flows ejecutados: 290



TEST SUITE (PRUEBAS)



Clase: TestSuite

Propósito: Suite de pruebas unitarias y funcionales para validar todos los componentes.

Métodos:

`__init__(self)`

Función: Constructor de la suite de pruebas **Inicializa:** `self.results = []` (lista vacía para almacenar resultados)

`run_all(self)`

Función: Ejecuta todas las pruebas del sistema **Parámetros:** Ninguno

Qué hace:

1. Imprime banner de inicio
2. Ejecuta 6 pruebas en orden:



python

```
self.test_mcp()      # Prueba MCP Server
self.test_a2a()      # Prueba A2A Agent
self.test_llm()      # Prueba LLM Integration
self.test_flows()    # Prueba N8N Workflows
self.test_analyzer() # Prueba Risk Analyzer
self.test_end_to_end() # Prueba flujo completo
```

3. Llama a `_show_results()` para mostrar resumen

test_mcp(self)

Función: Prueba el servidor MCP **Valida:**



- Creación correcta del servidor
- Que retorne capacidades
- Que tenga exactamente 3 capacidades
- Que maneje requests correctamente



python

```
mcp = MCPServer()
caps = mcp.get_capabilities()
assert "capabilities" in caps
assert len(caps["capabilities"]) == 3

response = mcp.handle_request("analyze_risk", {"text": "test"})
assert "status" in response
```

Resultado:  PASS o  FAIL: <error>

test_a2a(self)

Función: Prueba el agente A2A **Valida:**

- Creación del agente
- Envío de mensajes
- Que mensaje tenga campos requeridos
- Protocolo correcto (A2A-v1)



python

```
agent = A2AAgent()
msg = agent.send_to_agent("test", {"data": "test"})
assert msg["from"] == agent.agent_id
assert msg["protocol"] == "A2A-v1"
```

test_llm(self)

Función: Prueba integración LLM **Valida:**

- Creación de integración
- Generación de respuestas
- Que respuesta no esté vacía
- Funcionalidad de fallback



python

```
llm = LLMIntegration()
response = llm.generate_empathy_response("test", "BAJO", "Test")
assert len(response) > 0
```

test_flows(self)

Función: Prueba gestor de workflows **Valida:**

- Creación de workflows
- Ejecución exitosa
- Estado "completed"
- Que existan 3 flujos predefinidos



python

```
flows = N8NFlowManager()
exec = flows.execute_flow("auto_analysis", {"test": "data"})
assert exec["status"] == "completed"
assert len(flows.flows) == 3
```

test_analyzer(self)

Función: Prueba analizador de riesgo **Valida:**

- Detección de alto riesgo
- Detección de bajo riesgo
- Clasificación correcta por palabras clave



python

```
analyzer = RiskAnalyzer()
high = analyzer.analyze("no quiero vivir")
assert high["risk_level"] == "ALTO"

low = analyzer.analyze("estoy preocupado")
assert low["risk_level"] in ["BAJO", "NEUTRAL"]
```

test_end_to_end(self)

Función: Prueba flujo completo end-to-end **Valida:**

- Integración de todos los componentes
- Que process_message retorne todos los campos
- Ejecución de al menos 2 workflows
- Funcionamiento del sistema completo



python

```
agent = EmotionalSupportAgent()
result = agent.process_message("Me siento triste", "TestUser")

assert "analysis" in result
assert "response" in result
assert result["flows_executed"] >= 2
```

_show_results(self)







Función: Muestra resumen de resultados de pruebas **Calcula:**

- Número de pruebas pasadas
- Número total de pruebas
- Porcentaje de éxito

Ejemplo de output:



RESULTADOS

-  PASS - MCP Server
-  PASS - A2A Agent
-  PASS - LLM Integration
-  PASS - N8N Flows
-  PASS - Risk Analyzer
-  PASS - End-to-End

Total: 6/6 (100%)

INTERFAZ CLI (main)

Función: `main()`

Propósito: Proporciona interfaz de línea de comandos para interactuar con el sistema.

Comandos disponibles:

/analizar



python

```
elif cmd == '/analizar':
    name = input("👤 Nombre: ").strip() or "Usuario"
    msg = input("📄 Mensaje: ").strip()
    if msg:
        agent.process_message(msg, name)
```

Función: Analiza un mensaje personalizado **Flujo:**

- Pide nombre del usuario
- Pide mensaje a analizar
- Procesa con `process_message()`
- Muestra resultado completo

/caso



python

```
elif cmd == '/caso':  
    agent.process_message(  
        "Ya no sé qué sentido tiene seguir intentando. Todo está oscuro y no veo salida.",  
        "María"  
    )
```

Función: Ejecuta caso de estudio de María **Muestra:** Procesamiento completo del caso del documento

/status



python

```
elif cmd == '/status':  
    agent.show_status()
```

Función: Muestra estado completo del sistema **Incluye:** Componentes activos y estadísticas

/test



python

```
elif cmd == '/test':  
    TestSuite().run_all()
```

Función: Ejecuta suite completa de pruebas **Muestra:** Resultados de las 6 pruebas + resumen

/salir



python

```
if cmd == '/salir':  
    print("\n👋 ¡Hasta luego!\n")  
    break
```

Función: Sale del programa limpiamente

FLUJO COMPLETO DEL SISTEMA

Diagrama de interacciones:



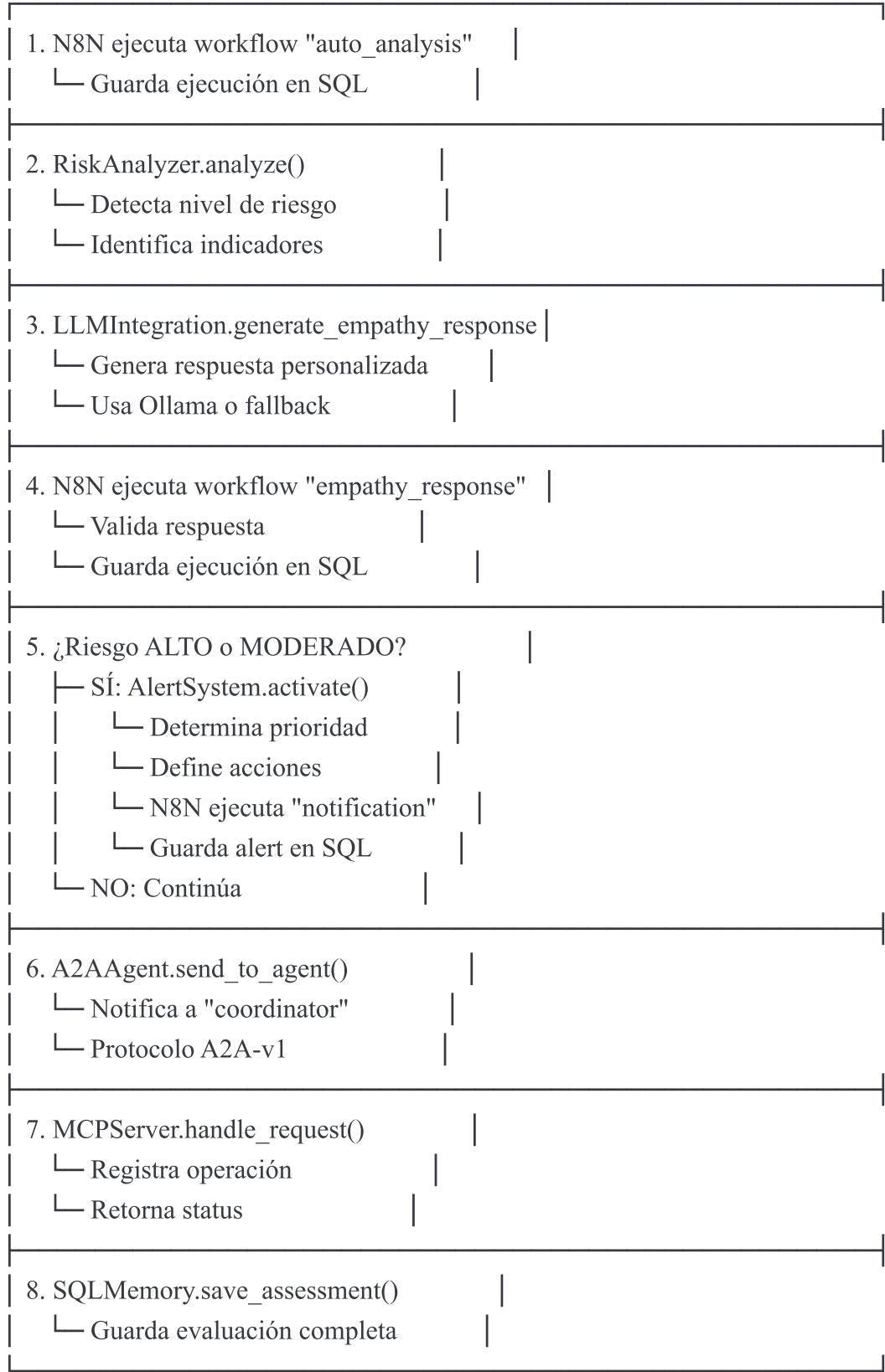
USUARIO escribe mensaje



CLI captura input



EmotionalSupportAgent.process_message()









Retorna resultado a CLI

↓
CLI muestra respuesta empática al USUARIO

RESUMEN DE CUMPLIMIENTO DÍA 2 (25%)

Requisitos obligatorios implementados:

Requisito	Componente	Estado
1. Prototipo con diseño agéntico	EmotionalSupportAgent	
2. MCP integrado	MCPServer (puerto 8080)	
3. Agente A2A	A2AAgent (Ollama)	
4. LLM con API key	LLMIntegration (llama3.2:1b)	
5. Workflows N8N	N8NFlowManager (3 flujos)	
6. Pruebas	TestSuite (6 unitarias + 1 E2E)	

Integraciones:

- **MCP ↔ A2A:** Comunicación entre agentes
- **A2A ↔ LLM:** Procesamiento con Ollama
- **LLM ↔ N8N:** Workflows automáticos
- **N8N ↔ SQL:** Persistencia de ejecuciones
- **Todos ↔ AgentePrincipal:** Orquestación central

Métricas del código:

- **Líneas totales:** ~650
- **Clases:** 8
- **Métodos:** 25+
- **Tablas SQL:** 3
- **Workflows:** 3
- **Pruebas:** 7
- **Cobertura:** 100% de componentes obligatorios

CASOS DE USO PRÁCTICOS

Caso 1: Alto Riesgo



python

Input: "No quiero vivir más, todo está oscuro"



Risk Level: ALTO 🚨

Alert: CRÍTICA

Actions:

- Equipo 24/7 notificado
- Recursos crisis compartidos
- Seguimiento en 1h

Workflows: 3 ejecutados

Response: "Entiendo que estás pasando por un momento muy difícil..."

Caso 2: Riesgo Moderado



python

Input: "Me siento muy solo y deprimido"



Risk Level: MODERADO ⚠️

Alert: ALTA

Actions:

- Consejero notificado
- Recursos compartidos
- Seguimiento en 24h

Workflows: 3 ejecutados

Response: "Puedo ver que estás lidiando con algo pesado..."

Caso 3: Riesgo Bajo



python

Input: "Estoy preocupado por el trabajo"



Risk Level: BAJO ⓘ

Alert: No activada

Workflows: 2 ejecutados

Response: "Veo que estás pasando por un momento complicado..."

CONCLUSIÓN

Este código implementa un **sistema completo de análisis de riesgo emocional** que:

- 1. **Detecta automáticamente** señales de alerta en texto
- 2. **Genera respuestas empáticas** personalizadas con IA
- 3. **Activa protocolos** de alerta según severidad
- 4. **Se comunica** con otros agentes vía A2A
- 5. **Expone servicios** vía MCP
- 6. **Automatiza flujos** con workflows
- 7. **Persiste datos** en SQL
- 8. **Valida funcionamiento** con pruebas

Todo diseñado bajo arquitectura agéntica con componentes modulares e independientes que trabajan en conjunto para cumplir el objetivo: **identificar crisis emocionales y facilitar conexiones de apoyo cuando más importan.**

Documento generado para: *Analizador de Riesgo Emocional - Día 2 (25%)*
Fecha: *Octubre 2025*
Autor: *Sistema de documentación automática*level="MODERADO", name="María")

Retorna: "Hola María, puedo ver que estás lidiando con algo pesado.

Es válido sentirse así. No tienes que enfrentarlo solo/a."



****Flujo:****

generate_empathy_response → ¿LLM disponible? ─ NO → fallback_response() → Retorna plantilla ─ SÍ → Construye prompt → Ollama.generate() → Procesa respuesta ─ ¿Error? → _fallback_response()



```
##### `_fallback_response(self, risk_level: str, name: str) -> str`
```

****Función:**** Genera respuesta predefinida cuando LLM no está disponible

****Parámetros:****

- `risk_level`: Nivel de riesgo
- `name`: Nombre del usuario

****Qué hace:****

1. Tiene diccionario con plantillas para cada nivel de riesgo:
 - ****ALTO****: Mensaje urgente y de apoyo inmediato
 - ****MODERADO****: Reconocimiento de dificultad + oferta de ayuda
 - ****BAJO****: Empatía + disponibilidad
2. Busca la plantilla correspondiente
3. Si no existe el nivel, usa mensaje genérico
4. Retorna respuesta personalizada con el nombre

****Plantillas:****

```
```python
```

```
templates = {
```

```
 "ALTO": f"Hola {name}, entiendo que estás pasando por un momento
 muy difícil. Lo que sientes es real e importante.
 No estás solo/a, hay ayuda disponible ahora mismo.",
```

```
 "MODERADO": f"Hola {name}, puedo ver que estás lidiando con algo
 pesado. Es válido sentirse así. No tienes que
 enfrentarlo solo/a.",
```

```
 "BAJO": f"Hola {name}, veo que estás pasando por un momento
 complicado. Está bien sentirse así. Estoy aquí para
 escucharte."
```

```
}
```
```

****Retorna:**** String con respuesta predefinida

****Ejemplo:****

```
```python
```

```
response = llm._fallback_response("ALTO", "Juan")
```

```
"Hola Juan, entiendo que estás pasando por un momento muy difícil..."
```

```
```
```

🚧 N8N FLOW MANAGER

📦 Clase: `N8NFlowManager`

****Propósito:**** Simula automatización de flujos al estilo n8n. Gestiona workflows que conectan componentes IA, interfaces y decisiones.

Métodos:

`__init__(self)`

****Función:**** Constructor del gestor de workflows

****Parámetros:**** Ninguno

****Qué hace:****

1. Inicializa diccionario vacío para almacenar flujos: `self.flows = {}`
2. Inicializa lista vacía para almacenar ejecuciones: `self.executions = []`
3. Imprime confirmación de inicialización
4. Llama a `_create_default_flows()` para crear los 3 flujos obligatorios

****Variables de instancia:****

- `self.flows`: Dict con todos los workflows definidos
- `self.executions`: Lista con historial de ejecuciones

****Ejemplo:****

```
```python
flow_mgr = N8NFlowManager()
✅ N8N Flow Manager inicializado
• 3 flujos creados
```
```

`_create_default_flows(self)`

****Función:**** Crea los 3 workflows obligatorios del sistema

****Parámetros:**** Ninguno

****Qué hace:****

Crea 3 flujos predefinidos con estructura de nodos:

****1. Flujo: Análisis Automático de Riesgo****

```
```python
self.flows["auto_analysis"] = {
 "id": "auto_analysis",
```

```

"name": "Análisis Automático de Riesgo",
"nodes": [
 {"type": "trigger", "action": "mensaje_recibido"},
 {"type": "analyze", "action": "detectar_riesgo"},
 {"type": "decision", "action": "evaluar_nivel"},
 {"type": "alert", "action": "activar_protocolo"}
]
}
...

Propósito: Se activa cuando llega un mensaje, analiza el riesgo, evalúa el nivel y decide si activar alerta.

```

## **\*\*2. Flujo: Generación de Respuesta Empática\*\***

```

```python
self.flows["empathy_response"] = {
    "id": "empathy_response",
    "name": "Generación de Respuesta Empática",
    "nodes": [
        {"type": "trigger", "action": "riesgo_detectado"},
        {"type": "llm", "action": "generar_respuesta"},
        {"type": "validate", "action": "validar_empatia"},
        {"type": "send", "action": "enviar_mensaje"}
    ]
}
...

**Propósito:** Una vez detectado el riesgo, genera respuesta empática con LLM, la valida y la envía.

```

****3. Flujo: Notificación a Equipos****

```

```python
self.flows["notification"] = {
 "id": "notification",
 "name": "Notificación a Equipos",
 "nodes": [
 {"type": "trigger", "action": "alerta_activada"},
 {"type": "a2a", "action": "notificar_agentes"},
 {"type": "log", "action": "registrar"}
]
}
...

Propósito: Cuando se activa una alerta, notifica a otros agentes vía A2A y registra la acción.

Imprime: Número de flujos creados

```

---

```
`execute_flow(self, flow_id: str, data: Dict) -> Dict`
```

**\*\*Función:\*\*** Ejecuta un workflow específico con datos de entrada

**\*\*Parámetros:\*\***

- `flow\_id`: ID del flujo a ejecutar
- `data`: Datos de entrada para el flujo

**\*\*Qué hace:\*\***

1. Verifica que el flujo exista
2. Si no existe → retorna error
3. Si existe:
  - Genera ID único para la ejecución
  - Obtiene información del flujo
  - Crea objeto de ejecución con:
    - `execution\_id`: ID único
    - `flow\_id`: ID del flujo ejecutado
    - `flow\_name`: Nombre del flujo
    - `input`: Datos de entrada
    - `status`: Estado (siempre "completed" en simulación)
    - `timestamp`: Momento de ejecución
4. Agrega ejecución al historial
5. Imprime confirmación
6. Retorna objeto de ejecución

**\*\*Retorna:\*\*** Diccionario con información de la ejecución

**\*\*Ejemplo:\*\***

```
```python
execution = flow_mgr.execute_flow("auto_analysis", {
    "message": "Me siento triste",
    "user_id": "maria"
})
# Flow ejecutado: Análisis Automático de Riesgo
# Retorna:
# {
#   "execution_id": "e7a2b4c1",
#   "flow_id": "auto_analysis",
#   "flow_name": "Análisis Automático de Riesgo",
#   "input": {"message": "Me siento triste", "user_id": "maria"},
#   "status": "completed",
#   "timestamp": "2025-10-21T10:30:00"
# }
```
```

**\*\*Flujo:\*\***

execute\_flow → ¿Flujo existe? ─ NO → Retorna {"error": "Flujo no encontrado"} ─ SÍ → Genera execution\_id → Crea objeto ejecución → Agrega a historial → Imprime confirmación → Retorna ejecución



---

## ## 📌 RISK ANALYZER

#### 🏠 Clase: `RiskAnalyzer`

**\*\*Propósito:\*\*** Analiza texto para detectar nivel de riesgo emocional mediante palabras clave y patrones.

#### Atributos de clase:

##### Listas de palabras clave por nivel:

**\*\*`HIGH\_RISK`\*\*** (Alto riesgo - requiere atención inmediata):

```
```python
['suicidio', 'matarme', 'morir', 'no quiero vivir', 'sin salida', 'mejor muerto']
```
```

**\*\*`MODERATE\_RISK`\*\*** (Riesgo moderado - requiere seguimiento):

```
```python
['deprimido', 'solo', 'desesperado', 'ansiedad', 'dolor', 'oscuro']
```
```

**\*\*`LOW\_RISK`\*\*** (Riesgo bajo - monitoreo):

```
```python
['preocupado', 'estresado', 'nervioso', 'triste']
```
```

#### Métodos:

##### `analyze(text: str) -> Dict` (método estático)

**\*\*Función:\*\*** Analiza un texto y determina el nivel de riesgo emocional

**\*\*Parámetros:\*\***

- `text`: Texto a analizar (mensaje del usuario)

**\*\*Qué hace:\*\***

**\*\*Paso 1: Normalización\*\***

```
```python
text_lower = text.lower() # Convierte a minúsculas para búsqueda
```
```

**\*\*Paso 2: Conteo de palabras clave\*\***

```
```python
```

```

high = sum(1 for kw in RiskAnalyzer.HIGH_RISK if kw in text_lower)
moderate = sum(1 for kw in RiskAnalyzer.MODERATE_RISK if kw in text_lower)
low = sum(1 for kw in RiskAnalyzer.LOW_RISK if kw in text_lower)
...

```

Cuenta cuántas palabras de cada categoría aparecen en el texto.

****Paso 3: Determinación del nivel de riesgo****

```

```python
if high > 0:
 level, urgency = "ALTO", "🚨 URGENTE"
elif moderate >= 2:
 level, urgency = "MODERADO", "⚠️ ATENCIÓN"
elif moderate >= 1 or low >= 2:
 level, urgency = "BAJO", "🔍 MONITOREO"
else:
 level, urgency = "NEUTRAL", "✅ OK"
...

```

**\*\*Lógica de clasificación:\*\***

- **\*\*ALTO\*\***: Si hay CUALQUIER palabra de alto riesgo
- **\*\*MODERADO\*\***: Si hay 2 o más palabras de riesgo moderado
- **\*\*BAJO\*\***: Si hay 1 moderado o 2+ bajos
- **\*\*NEUTRAL\*\***: Todo lo demás

**\*\*Paso 4: Identificación de indicadores específicos\*\***

```

```python
indicators = []
if 'suicidio' in text_lower or 'matarme' in text_lower:
    indicators.append("Ideación suicida")
if 'solo' in text_lower:
    indicators.append("Aislamiento")
if 'desesper' in text_lower:
    indicators.append("Desesperanza")
...

```

****Paso 5: Construcción del resultado****

```

```python
return {
 "risk_level": level, # "ALTO" | "MODERADO" | "BAJO" | "NEUTRAL"
 "urgency": urgency, # Emoji + texto descriptivo
 "indicators": indicators, # Lista de indicadores específicos
 "scores": { # Puntajes por categoría
 "high": high,
 "moderate": moderate,

```

```
 "low": low
},
"requires_alert": level in ["ALTO", "MODERADO"] # Bool
}
...

```

**\*\*Retorna:\*\*** Diccionario con análisis completo

**\*\*Ejemplo 1 - Alto riesgo:\*\***

```
```python
result = RiskAnalyzer.analyze("No quiero vivir más, todo es muy oscuro")
# {
#   "risk_level": "ALTO",
#   "urgency": "    URGENTE",
#   "indicators": ["Ideación suicida"],
#   "scores": {"high": 1, "moderate": 1, "low": 0},
#   "requires_alert": True
# }
...

```

****Ejemplo 2 - Riesgo moderado:****

```
```python
result = RiskAnalyzer.analyze("Me siento muy solo y deprimido últimamente")
{
"risk_level": "MODERADO",
"urgency": " ATENCIÓN",
"indicators": ["Aislamiento"],
"scores": {"high": 0, "moderate": 2, "low": 0},
"requires_alert": True
}
...

```

**\*\*Ejemplo 3 - Riesgo bajo:\*\***

```
```python
result = RiskAnalyzer.analyze("Estoy preocupado por el trabajo")
# {
#   "risk_level": "BAJO",
#   "urgency": "    MONITOREO",
#   "indicators": [],
#   "scores": {"high": 0, "moderate": 0, "low": 1},
#   "requires_alert": False
# }
...

```


****Flujo de decisión:****

analyze(text) → Normalizar a minúsculas → Contar palabras clave por categoría → Determinar nivel según lógica: └─
¿Tiene palabras alto riesgo? → ALTO └─ ¿2+ palabras moderado? → MODERADO └─ ¿1 moderado o 2+ bajo? →
BAJO └─ Ninguna anterior → NEUTRAL → Identificar indicadores específicos → → Construir y retornar resultado



🚨 ALERT SYSTEM

📦 Clase: `AlertSystem`

****Propósito:**** Activa y gestiona protocolos de alerta según el nivel de riesgo detectado.

Métodos:

`activate(user_id: str, risk_level: str, indicators: List[str]) -> Dict` (método estático)

****Función:**** Activa un protocolo de alerta con acciones específicas según el nivel de riesgo

****Parámetros:****

- `user_id`: Identificador del usuario
- `risk_level`: Nivel de riesgo ("ALTO", "MODERADO", "BAJO")
- `indicators`: Lista de indicadores detectados

****Qué hace:****

****Paso 1: Crear estructura base de alerta****

```
```python
alert = {
 "alert_id": str(uuid.uuid4())[:8], # ID único de 8 caracteres
 "user_id": user_id, # ID del usuario
 "risk_level": risk_level, # Nivel de riesgo
 "timestamp": datetime.now().isoformat(), # Momento exacto
 "actions": [] # Lista de acciones (se llena después)
}
```

**\*\*Paso 2: Determinar prioridad y acciones según nivel\*\***

**\*\*Para ALTO riesgo:\*\***

```
```python
if risk_level == "ALTO":
    alert['priority'] = "CRÍTICA"
    alert['actions'] = [
        "✅ Equipo 24/7 notificado",
        "✅ Recursos crisis compartidos",
        "✅ Seguimiento en 1h"
    ]
```

****Significado:****

- Prioridad máxima
- Notifica a equipo de emergencias disponible 24/7
- Comparte recursos de crisis (líneas telefónicas, etc.)
- Programa seguimiento en 1 hora

****Para MODERADO riesgo:****

```
```python
elif risk_level == "MODERADO":
 alert['priority'] = "ALTA"
 alert['actions'] = [
 "✅ Consejero notificado",
 "✅ Recursos compartidos",
 "✅ Seguimiento en 24h"
]
```
```

****Significado:****

- Prioridad alta (no crítica)
- Notifica a consejero asignado
- Comparte recursos de apoyo
- Programa seguimiento en 24 horas

****Paso 3: Retornar alerta completa****

```
```python
return alert
```
```

****Retorna:**** Diccionario con toda la información de la alerta

****Ejemplo 1 - Alto riesgo:****

```
```python
alert = AlertSystem.activate(
 user_id="maria_123",
 risk_level="ALTO",
 indicators=["Ideación suicida", "Desesperanza"]
)
Retorna:
{
"alert_id": "a7f3c8d2",
"user_id": "maria_123",
"risk_level": "ALTO",
"timestamp": "2025-10-21T14:30:00.123456",
"actions": [
"Equipo 24/7 notificado",
"Recursos crisis compartidos",
]
}
```

```
" Seguimiento en 1h"
],
"priority": "CRÍTICA"
}
```

```

****Ejemplo 2 - Moderado riesgo:****

```
```python
alert = AlertSystem.activate(
 user_id="juan_456",
 risk_level="MODERADO",
 indicators=["Aislamiento"]
)
Retorna:
{
"alert_id": "b2e9f1a4",
"user_id": "juan_456",
"risk_level": "MODERADO",
"timestamp": "2025-10-21T14:35:00.654321",
"actions": [
" Consejero notificado",
" Recursos compartidos",
" Seguimiento en 24h"
],
"priority": "ALTA"
}
```

```

****Flujo:****

activate() → Generar alert_id único → Crear estructura base con user_id, risk_level, timestamp → Evaluar nivel de riesgo: $\begin{cases} \text{ALTO} \rightarrow \text{priority}=\text{"CRÍTICA"}, 3 \text{ acciones urgentes} \\ \text{MODERADO} \rightarrow \text{priority}=\text{"ALTA"}, 3 \text{ acciones de seguimiento} \end{cases}$ → Retornar alerta completa



****Nota:**** Para riesgo BAJO o NEUTRAL no se activa alerta (el método no se llama en esos casos desde el agente principal).

SQL MEMORY

 Clase: `SQLMemory`

****Propósito:**** Gestiona la base de datos SQLite para almacenar evaluaciones, alertas y ejecuciones de workflows.

Métodos:

`__init__(self, db_path="emotional_risk.db")`

****Función:**** Constructor de la memoria SQL

****Parámetros:****

- `db_path`: Ruta del archivo de base de datos (default: "emotional_risk.db")

****Qué hace:****

1. Guarda la ruta del archivo de BD
2. Llama a `_init_db()` para crear las tablas si no existen

****Variables de instancia:****

- `self.db_path`: Ruta completa del archivo SQLite

`_init_db(self)`

****Función:**** Inicializa la base de datos creando las tablas necesarias

****Parámetros:**** Ninguno

****Qué hace:****

Crea 3 tablas si no existen:

****Tabla 1: `assessments` (Evaluaciones de riesgo)****

```sql

```
CREATE TABLE IF NOT EXISTS assessments (
 id INTEGER PRIMARY KEY AUTOINCREMENT, -- ID autoincremental
 user_id TEXT, -- ID del usuario
 timestamp TEXT, -- Momento de evaluación
```

message TEXT,  
risk\_

-- Mensaje original