

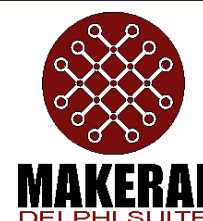
MakerAi 2.5 AGENTS Graph

CimaMaker

Correo electrónico: gustavoeenriquez@gmail.com

Sitio web: <https://makerai.cimamaker.com>

Doc. Versión 1.0



CONTENIDO

Módulo 1: ¿Por qué necesitamos orquestrar la IA? Más allá de la simple pregunta y respuesta.	3
La Revolución Silenciosa en tu IDE	3
El Problema: El LLM es un Cerebro sin Manos.....	3
La Solución: Agentes y Grafos de IA	4
Módulo 2: Anatomía de un Grafo de IA con MakerAI Agents	6
La Metáfora: Un Diagrama de Flujo Inteligente.....	6
2.1 El Orquestador (TAIAgents): El Cerebro del Operativo.....	6
2.2 El Nodo (TAIAgentsNode): La Unidad de Trabajo	7
Propiedades y Conceptos Clave:	7
El Papel del Nodo en el Grafo:	9
2.3 El Enlace (TAIAgentsLink): Definiendo el Flujo y la Lógica.....	9
Capacidades Clave de un Enlace:.....	9
2.4 La Pizarra (TAIBlackboard): La Memoria Compartida del Grafo	11
¿Qué es y cómo funciona?	11
Métodos Principales:	11
2.5 La Herramienta (TAiToolBase): Dando Capacidades al Agente	13
¿Por qué usar Herramientas?	13
Anatomía de una Herramienta:.....	13
¿Cómo se usa?	14

MÓDULO 1: ¿POR QUÉ NECESITAMOS ORQUESTAR LA IA? MÁS ALLÁ DE LA SIMPLE PREGUNTA Y RESPUESTA.

La Revolución Silenciosa en tu IDE

Como desarrollador Delphi, estás acostumbrado a construir aplicaciones robustas, rápidas y fiables. Durante décadas, hemos dominado la creación de sistemas complejos, desde ERPs y puntos de venta hasta software industrial. Hoy, una nueva revolución está llamando a nuestra puerta: la Inteligencia Artificial Generativa y los Modelos de Lenguaje Grandes (LLMs) como GPT, Gemini o Llama.

Integrar una llamada a una API de un LLM en una aplicación Delphi es relativamente sencillo. Con componentes como TRESTClient, podemos enviar una pregunta y recibir una respuesta. Esto es útil, pero es solo la punta del iceberg. Pronto nos encontramos con preguntas más complejas:

- *"¿Cómo puedo hacer que el LLM consulte mi base de datos de clientes para responder una pregunta?"*
- *"¿Y si necesito que, después de generar un informe, lo envíe automáticamente por correo electrónico?"*
- *"¿Cómo puedo mantener una conversación coherente con un usuario, recordando lo que se dijo antes?"*
- *"¿Qué pasa si una tarea requiere múltiples pasos de razonamiento y acceso a diferentes herramientas?"*

Aquí es donde una simple llamada a la API se queda corta.

El Problema: El LLM es un Cerebro sin Manos

Imagina un LLM como un cerebro increíblemente inteligente y elocuente, pero que está aislado en una habitación sin puertas ni ventanas. Puede razonar, escribir y generar ideas, pero no puede interactuar con el mundo exterior. No puede buscar en internet, no puede leer un archivo en tu disco duro, no puede consultar tu base de datos SQL, ni puede ejecutar una acción en tu aplicación.

Cuando intentamos resolver problemas del mundo real que requieren más que solo generar texto, nos enfrentamos a estas limitaciones:

1. **Falta de Contexto Actualizado:** Los LLMs son entrenados con datos que tienen una fecha de corte. No saben qué pasó ayer, ni cuál es el stock actual de tu producto más vendido.

2. **Incapacidad para Actuar:** No pueden ejecutar acciones. No pueden hacer clic en un botón, llamar a una API externa (por sí mismos) o modificar un registro en una base de datos.
3. **Complejidad de Múltiples Pasos:** Tareas como "Investiga las últimas tendencias del mercado para el producto X, redacta un borrador de correo para el equipo de marketing y guárdalo en la carpeta de borradores" son imposibles de realizar en un solo paso.

Intentar gestionar esta lógica manualmente en tu código Delphi puede convertirse rápidamente en un "espagueti" de if-then-else, llamadas anidadas a APIs y una gestión de estado frágil y difícil de mantener.

La Solución: Agentes y Grafos de IA

La solución no es pedirle al LLM que haga todo, sino tratarlo como un componente más (aunque muy especial) dentro de un flujo de trabajo más grande. Aquí es donde entra en juego la **orquestación**.

MakerAI Agents te permite construir estos flujos de trabajo inteligentes de una manera estructurada, visual y potente. La idea central es simple: **descomponer un problema complejo en una serie de pasos interconectados**, formando lo que llamamos un "Grafo de IA".

Para entender cómo funciona, introduzcamos algunos conceptos clave:

- **Agente (Agent):** Un agente no es solo un LLM. Es un sistema que utiliza un LLM como su "motor de razonamiento" para tomar decisiones y actuar. El agente observa una entrada (por ejemplo, una pregunta del usuario), piensa qué pasos debe seguir para resolverla, y utiliza las herramientas a su disposición para ejecutar esos pasos.
- **Herramienta (Tool):** Una herramienta es la "mano" que le damos a nuestro cerebro de IA. Es una función o procedimiento Delphi bien definido que el agente puede ejecutar para interactuar con el mundo exterior.
 - **TDatabaseTool:** Una herramienta para ejecutar consultas SQL.
 - **TFileTool:** Una herramienta para leer o escribir archivos.
 - **TWebApiTool:** Una herramienta para consumir datos de una API externa (por ejemplo, el tiempo, precios de acciones, etc.).
- **Cadenas (Chains) vs. Grafos (Graphs):**
 - Una **Cadena** es la forma más simple de orquestación: un flujo lineal donde la salida del paso A se convierte en la entrada del paso B, la de B en la de C, y así sucesivamente. *Ejemplo: Traducir un texto -> Resumir el texto traducido -> Analizar el sentimiento del resumen.*

- Un **Grafo** es una estructura mucho más poderosa y flexible, necesaria para la mayoría de los problemas del mundo real. Permite:
 - **Ramas (Branches)**: Tomar diferentes caminos basados en una condición (ej. si el sentimiento es positivo, hacer X; si es negativo, hacer Y).
 - **Paralelización (Fan-out)**: Ejecutar múltiples pasos al mismo tiempo para mejorar la eficiencia.
 - **Uniones (Joins)**: Esperar a que varias ramas paralelas terminen antes de continuar.
 - **Ciclos (Cycles)**: Repetir un conjunto de pasos hasta que se cumpla una condición (ej. el agente intenta usar una herramienta, si falla, razona de nuevo y lo intenta de otra forma).
- **Estado (State) y la Pizarra (Blackboard)**: Para que los diferentes pasos de un grafo se comuniquen entre sí, necesitan un lugar compartido donde puedan leer y escribir información. Este es el rol de la **Pizarra (TAI Blackboard)**. Es una memoria persistente para una única ejecución del grafo. Un nodo puede escribir un resultado en la pizarra (ej. `Blackboard.SetString('CustomerID', '123')`), y otro nodo, más adelante en el flujo, puede leer ese valor para realizar su tarea.

MÓDULO 2: ANATOMÍA DE UN GRAFO DE IA CON MAKERAI AGENTS

La Metáfora: Un Diagrama de Flujo Inteligente

Si alguna vez has diseñado un diagrama de flujo para representar un proceso de negocio o un algoritmo, ya estás a mitad de camino para entender cómo funciona **MakerAI Agents**. Imagina cada caja de tu diagrama como una unidad de trabajo (TAIAgentsNode) y cada flecha que las conecta como una ruta de flujo (TAIAgentsLink). Ahora, imagina que dentro de algunas de esas cajas hay un "cerebro" de IA capaz de tomar decisiones, y que las flechas pueden cambiar de dirección dinámicamente según los resultados. Eso es, en esencia, un Grafo de IA.

MakerAI Agents te permite construir estos diagramas de flujo inteligentes directamente en el diseñador de formularios de Delphi o crearlos dinámicamente por código. Cada pieza del puzzle es un componente Delphi, con propiedades que puedes configurar en el Inspector de Objetos y eventos que puedes implementar en el Editor de Código.

Vamos a desglosar las piezas fundamentales que componen cualquier grafo que construyas.

2.1 El Orquestador (TAIAgents): El Cerebro del Operativo

El componente TAIAGents es el corazón y el cerebro de todo el sistema. No realiza ninguna tarea de negocio por sí mismo, pero es responsable de gestionar y ejecutar el grafo completo. Piensa en él como el director de una orquesta: no toca ningún instrumento, pero se asegura de que todos los músicos (los nodos) entren a tiempo, sigan la partitura (el flujo) y trabajen en armonía.

Colocarás un componente TAIAGents en tu formulario o DataModule como punto de entrada principal para tu lógica de IA.

Responsabilidades Clave:

1. **Contenedor del Grafo:** Es el "dueño" de todos los nodos (TAIAgentsNode) y enlaces (TAIAgentsLink) que forman parte del grafo. Cuando colocas un nodo en el formulario, se asocia automáticamente con el orquestador principal.
2. **Punto de Entrada y Salida:** A través de sus propiedades StartNode y EndNode, le dices al orquestador dónde debe comenzar y dónde debe terminar el flujo de ejecución.
 - StartNode: El primer nodo que se ejecutará cuando llames al método Run.
 - EndNode: Un nodo especial que marca la finalización exitosa del grafo.

3. **Motor de Ejecución:** El método `Run(Input: string): ITask` es el que pone todo en marcha. Inicia la ejecución del grafo en un hilo secundario (`TTask`) para no bloquear la interfaz de usuario de tu aplicación, y devuelve una tarea (`ITask`) que puedes usar para esperar a que finalice si es necesario.
4. **Gestión de Estado Centralizada:** El orquestador posee la **Pizarra (Blackboard)**, que es el estado compartido para toda la ejecución del grafo. Veremos la pizarra en detalle más adelante, pero es crucial entender que es el `TAIAgents` quien la gestiona.
5. **Manejo de Eventos Globales:** Proporciona eventos a nivel de grafo para gestionar la comunicación y los errores de forma centralizada.
 - `OnPrint`: Un evento útil para el *logging* y la depuración. Cualquier nodo o enlace en el grafo puede llamar a su método `Print`, y este evento se disparará.
 - `OnError`: Un "try-catch" global para tu grafo. Si cualquier nodo o enlace lanza una excepción durante la ejecución, este evento se captura, permitiéndote registrar el error, notificar al usuario o decidir si abortar la ejecución.
 - `OnEnd`: Se dispara cuando el grafo finaliza su ejecución (ya sea llegando al `EndNode` o siendo abortado). Aquí es donde normalmente recogerás el resultado final del `Output` del último nodo o del `Blackboard`.
 - `OnConfirm`: Facilita la interacción con el usuario. Permite que un nodo detenga el flujo y haga una pregunta al usuario (a través de la UI principal), esperando una respuesta para poder continuar.

En la Práctica:

En tu formulario, tendrás un único `TAIAgents`. Lo configurarás asignando los nodos de inicio y fin, y probablemente implementarás los eventos `OnEnd` y `OnError` para saber cuándo ha terminado el trabajo y si algo ha salido mal. Toda la lógica compleja y específica de cada paso residirá en los nodos individuales, manteniendo tu código limpio y organizado.

2.2 El Nodo (`TAIAgentsNode`): La Unidad de Trabajo

Si el `TAIAgents` es el director de la orquesta, el `TAIAgentsNode` es el músico. Cada nodo representa un **paso discreto y específico** dentro de tu flujo de trabajo. Es una caja en nuestro "diagrama de flujo inteligente" que recibe una entrada, realiza una acción y produce una salida.

La belleza de este enfoque es que puedes crear nodos para cualquier tipo de tarea, desde una simple manipulación de texto hasta una compleja llamada a un modelo de IA, una consulta a base de datos o la ejecución de una herramienta especializada.

Propiedades y Conceptos Clave:

- **Input y Output (String):**

- Input: Los datos que recibe el nodo para trabajar. Por defecto, el Input de un nodo se rellena automáticamente con el Output del nodo que lo precedió en el flujo.
- Output: El resultado del trabajo realizado por el nodo. Este valor será pasado como Input al siguiente nodo conectado. Si no se modifica, el Output es igual al Input, permitiendo que los datos fluyan a través de nodos que no necesitan alterarlos.
- **OnExecute (Evento):**
Este es el corazón lógico del nodo. Es un evento de tipo procedure(Node, BeforeNode: TAIAgentsNode; Link: TAIAgentsLink; Input: String; var Output: String) donde escribes el código que define lo que hace este nodo.
 - Recibes el Input.
 - Realizas tu lógica (llamar a una función, una API, procesar el texto...).
 - Asignas el resultado al parámetro var Output.
- **Next (TAIAgentsLink):**
Esta propiedad conecta el nodo con el siguiente paso del flujo. Desde un nodo, solo puede salir **un enlace**. Sin embargo, como veremos en la siguiente sección, ese único enlace puede tener múltiples destinos, permitiendo la ejecución en paralelo.
- **Tool (TAiToolBase):**
Una alternativa potente al evento OnExecute. En lugar de escribir el código directamente en el evento del formulario, puedes asociar al nodo una "Herramienta" predefinida (un componente TAiToolBase). Cuando el nodo se ejecuta, si tiene una Tool asignada, llamará al método Execute de esa herramienta. Esto promueve la **reutilización de código** y la **separación de responsabilidades**. Por ejemplo, puedes tener una única TDatabaseQueryTool y usarla en múltiples nodos a lo largo de diferentes grafos. El nodo que tiene la máxima prioridad es el OnExecute por sobre el Tool.
- **JoinMode y Sincronización:**
Un nodo puede ser el punto de llegada de múltiples enlaces (por ejemplo, después de una ejecución en paralelo). La propiedad JoinMode controla cómo se comporta el nodo en esta situación:
 - jmAny (Por defecto): El nodo se ejecutará **cada vez** que una de las ramas de entrada llegue a él. Esto es ideal para grafos que contienen ciclos o bucles.
 - jmAll: El nodo **esperará** a que **todas** las ramas de entrada hayan completado su ejecución y llegado a él antes de ejecutarse una sola vez. Esto es esencial para sincronizar tareas paralelas antes de proceder al siguiente paso (como unificar los resultados en un informe final).

El Papel del Nodo en el Grafo:

Puedes tener diferentes tipos de nodos según su función:

- **Nodos de Procesamiento:** Realizan una tarea específica, como llamar a un LLM, formatear texto, realizar un cálculo, etc.
- **Nodos de Enrutamiento:** Nodos cuya principal función es analizar el estado actual (a menudo leyendo del Blackboard) para tomar una decisión que influirá en el camino que tomará el flujo.
- **Nodos de Herramienta:** Nodos que actúan como ejecutores de acciones en el mundo real, utilizando el componente Tool.
- **Nodos de Espera/Unión:** Nodos configurados con JoinMode = jmAll que sirven como puntos de sincronización en el grafo.

Al diseñar tu grafo, piensa en cada tarea como un nodo independiente. Esto hace que tu flujo de trabajo sea modular, más fácil de entender, depurar y mantener.

2.3 El Enlace (TAIAgentsLink): Definiendo el Flujo y la Lógica

Si los nodos son las ciudades en nuestro mapa, los enlaces (TAIAgentsLink) son las carreteras que las conectan. Un enlace define la ruta que siguen los datos y el control de un nodo a otro. Sin embargo, en **MakerAI Agents**, un enlace es mucho más que una simple flecha; puede contener su propia lógica para crear flujos de trabajo dinámicos y complejos. Cada TAIAgentsNode tiene una única propiedad de salida, Next, donde se asigna un TAIAgentsLink. Es este enlace el que determina qué sucede a continuación.

Capacidades Clave de un Enlace:

1. Conexión Simple (Destino Único):

En su forma más básica, un enlace conecta un nodo con otro. Esto se hace asignando el nodo de destino a la propiedad NextA del enlace. Este es el pilar para construir cadenas lineales (A -> B -> C).

2. Ejecución en Paralelo (Fan-out):

Aquí es donde el poder comienza a manifestarse. Un único enlace puede tener múltiples destinos. Las propiedades NextA, NextB, NextC, y NextD pueden apuntar cada una a un nodo diferente. Cuando el flujo llega a este enlace, **todos los nodos de destino se ejecutan simultáneamente**, cada uno en su propio hilo. Esto es ideal para tareas que no dependen entre sí y pueden realizarse al mismo tiempo para ahorrar tiempo.

- *Ejemplo:* Un nodo recibe una solicitud de producto. Su enlace de salida se divide para que tres nodos se ejecuten en paralelo: uno consulta el stock, otro busca opiniones de clientes y un tercero busca productos competidores.

3. Rutas de Fallo (Fallback):

La propiedad NextNo define una ruta alternativa. Se utiliza en conjunto con la lógica condicional del propio enlace para manejar casos de "fallo" o cuando una condición no se cumple.

- MaxCicles: Esta propiedad establece un número máximo de veces que se puede pasar por este enlace en un bucle antes de considerarlo un fallo y tomar la ruta NextNo. Esto es un mecanismo de seguridad crucial para evitar bucles infinitos.

4. Enrutamiento Condicional (Decisión Inteligente):

Esta es una de las características más avanzadas y potentes, que eleva tu grafo de un simple script a un sistema de razonamiento. Un enlace puede funcionar como un "switch" o un "router".

- **Cómo funciona:** En lugar de usar NextA, NextB, etc., se define un diccionario de destinos condicionales. El enlace, al ejecutarse, consultará una clave específica en el Blackboard (por ejemplo, Blackboard.GetString('next_route')). El valor de esa clave determinará a cuál de los nodos de destino se enrutará el flujo.
- *Ejemplo:* Un nodo (ClassifierNode) analiza un correo electrónico y escribe en el Blackboard si es "VENTAS", "SOPORTE" o "SPAM". El enlace de salida de este nodo está configurado con tres rutas condicionales. Si 'next_route' es "VENTAS", el flujo va al SalesTeamNode; si es "SOPORTE", va al SupportTeamNode, y así sucesivamente.

5. Lógica Personalizada en la Transición (OnExecute):

Al igual que un nodo, un enlace también tiene un evento OnExecute. Este evento te da un control preciso sobre la transición.

procedure(Node: TAIAgentsNode; Link: TAIAgentsLink; var IsOk: Boolean; var Handled: Boolean)

- Puedes realizar validaciones antes de pasar al siguiente nodo.
- Modificando IsOk := False, puedes forzar que la transición se considere un "fallo", lo que puede redirigir el flujo a NextNo si está configurado.
- Estableciendo Handled := True, puedes detener completamente el flujo en este punto, sin que el enlace continúe hacia ninguno de sus destinos.

En esencia, el TAIAgentsLink te permite decidir no solo **hacia dónde** va el flujo, sino también **cómo** y **por qué** va allí. La combinación de paralelismo, rutas de fallo y enrutamiento condicional es lo que te permite construir grafos que pueden adaptarse, reaccionar y tomar decisiones complejas.

2.4 La Pizarra (TAIBlackboard): La Memoria Compartida del Grafo

Si los nodos son las oficinas de una empresa y los enlaces son los pasillos, la pizarra (TAIBlackboard) es el tablón de anuncios central donde todos pueden leer mensajes importantes y dejar notas para los demás.

En un grafo, los nodos se ejecutan a menudo de forma independiente y, en casos de paralelismo, incluso en hilos diferentes. El Blackboard es el mecanismo que les permite comunicarse y compartir información de manera segura y centralizada. Es, en efecto, la **memoria a corto plazo** de una única ejecución del grafo.

El Blackboard es una propiedad del orquestador TAIAgents, lo que significa que solo hay una instancia por grafo y es accesible desde cualquier nodo, enlace o herramienta dentro de ese grafo.

¿Qué es y cómo funciona?

Técnicamente, el TAIBlackboard es un diccionario seguro para hilos (thread-safe) que almacena pares de clave-valor. La clave es siempre un string, y el valor puede ser de varios tipos (TValue), incluyendo:

- String
- Integer
- Boolean
- Double
- Y otros tipos soportados por TValue.

Métodos Principales:

Para facilitar su uso, el Blackboard ofrece métodos directos y tipados:

- SetString(const AKey, AValue: string): Escribe o actualiza un valor de tipo texto.
- GetString(const AKey: string; const ADefault: string = ""): string: Lee un valor de texto.
Si la clave no existe, devuelve el valor por defecto.
- SetInteger(...), GetInteger(...)
- SetBoolean(...), GetBoolean(...)

Y métodos genéricos para trabajar con TValue:

- SetValue(const AKey: string; const AValue: TValue)
- TryGetValue(const AKey: string; out AValue: TValue): Boolean

Casos de Uso Cruciales:

1. Pasar Información Compleja entre Nodos:

Mientras que el Input/Output de un nodo es ideal para pasar el resultado principal de un paso al siguiente, a menudo necesitas pasar múltiples piezas de información.

- *Ejemplo:* Un GetUserNode obtiene el ID, el nombre y el nivel de suscripción de un usuario. En lugar de concatenar todo en un solo string de salida, puede escribir cada dato en el Blackboard:

```
Graph.Blackboard.SetInteger('UserID', 123);
Graph.Blackboard.SetString('UserName', 'John Doe');
Graph.Blackboard.SetString('Subscription', 'Premium');
```

Nodos posteriores pueden ahora leer esta información directamente, sin necesidad de parsear un string complejo.

2. Habilitar el Enrutamiento Condicional:

Este es el caso de uso más importante. Como vimos en la sección de enlaces, el enrutamiento condicional depende de que un nodo escriba una "decisión" en el Blackboard.

○ Ejemplo:

```
// Dentro del OnExecute de un nodo de clasificación
var Classification: string;
Classification := MyLLM.Classify(Input); // Devuelve 'support' o 'sales'
Graph.Blackboard.SetString('next_route', Classification);
```

El siguiente enlace en el grafo leerá el valor de 'next_route' para decidir a qué nodo dirigir el flujo.

3. Agregar Resultados de Ramas Paralelas:

Cuando tienes múltiples nodos ejecutándose en paralelo, cada uno puede realizar su trabajo y escribir su resultado en el Blackboard usando una clave única. El nodo de unión (JoinNode) que viene después puede entonces leer todos los resultados de la pizarra para consolidarlos.

○ Ejemplo:

```
// En SummarizeNode
Graph.Blackboard.SetString('summary_result', SummaryText);
// En KeywordsNode
Graph.Blackboard.SetString('keywords_result', KeywordsList);
// En ReportNode (el nodo de unión)
var FinalReport: TStringBuilder;
FinalReport.AppendLine(Graph.Blackboard.GetString('summary_result'));
FinalReport.AppendLine(Graph.Blackboard.GetString('keywords_result'));
```

4. Mantener Contadores o Banderas de Estado:

Puedes usar el Blackboard para rastrear el estado de la ejecución, como el número de reintentos, si una acción crítica ya se realizó, etc.

El Blackboard se limpia automáticamente al inicio de cada ejecución (Run), asegurando que cada operación del grafo comience con un estado limpio. Es la herramienta que transforma una colección de pasos aislados en un proceso coherente y con estado.

2.5 La Herramienta (TAiToolBase): Dando Capacidades al Agente

Hasta ahora, hemos visto cómo construir flujos de trabajo internos. Pero el verdadero poder de un agente de IA reside en su capacidad para **interactuar con sistemas externos**: consultar una base de datos, leer un archivo, llamar a una API web o ejecutar cualquier otra acción en tu aplicación.

Aquí es donde entra en juego la **Herramienta (TAiToolBase)**.

Una herramienta es un componente Delphi reutilizable que encapsula una capacidad específica. En lugar de escribir la lógica de acceso a datos o a una API directamente en el evento OnExecute de un nodo, la encapsulas dentro de una clase que hereda de TAiToolBase.

¿Por qué usar Herramientas?

1. **Reutilización:** Puedes crear una herramienta una vez (por ejemplo, TExecuteSQLQueryTool) y usarla en innumerables nodos a lo largo de muchos grafos diferentes. Si necesitas actualizar la forma en que te conectas a la base de datos, lo haces en un solo lugar: la clase de la herramienta.
2. **Separación de Responsabilidades (SoC):** Mantiene tu grafo "limpio". El grafo se encarga de la *orquestación* (el "qué" y el "cuándo"), mientras que las herramientas se encargan de la *ejecución* (el "cómo"). Esto hace que tu lógica sea mucho más fácil de leer, probar y mantener.
3. **Abstracción:** Un nodo no necesita saber los detalles de cómo funciona una herramienta. Simplemente sabe que tiene una herramienta que, por ejemplo, "obtiene el precio de un producto". La complejidad de la llamada a la API, el manejo de errores y el parseo de la respuesta está oculta dentro de la herramienta.
4. **Preparación para Agentes Autónomos:** Este es el punto más importante de cara al futuro. Los sistemas de agentes avanzados (como los que usan *Function Calling* o *ReAct*) funcionan presentando a un LLM una lista de herramientas disponibles. El LLM, basándose en la pregunta del usuario, decide qué herramienta usar y con qué parámetros. Al estructurar tus capacidades como TAiToolBase, ya estás construyendo el sistema de la manera correcta para poder, en el futuro, permitir que el LLM tome estas decisiones de forma autónoma.

Anatomía de una Herramienta:

Crear una herramienta es muy sencillo. Debes crear una nueva clase que herede de TAiToolBase y sobrescribir un único método:

```
type
  TMyCustomTool = class(TAiToolBase)
  protected
```

```

    procedure Execute(ANode: TAIAgentsNode; const AInput: string; var AOutput: string);
    override;
    published
        property Description; // Heredada y muy importante
    end;

procedure TMyCustomTool.Execute(ANode: TAIAgentsNode; const AInput: string; var AOutput:
string);
begin
    // Aquí- va tu lógica.
    // AInput: Contiene la entrada del nodo que la está usando.
    // ANode: Te da acceso al nodo que te está ejecutando, por si necesitas
    //         llamar a Print() o acceder al Blackboard (ANode.Graph.Blackboard).

    // Ejemplo:
    ANode.Print(Format('Mi herramienta se está ejecutando con la entrada: %s', [AInput]));
    AOutput := 'Resultado de mi herramienta procesando ' + AInput;
end;

```

- **Execute (Método):** Es el corazón de la herramienta. Aquí implementas lo que la herramienta debe hacer.
- **Description (Propiedad):** Es fundamental documentar qué hace la herramienta aquí. Esta descripción no es solo para otros desarrolladores; es el texto que un LLM usaría para entender la capacidad de la herramienta y decidir si debe usarla. Una buena descripción sería: "Busca el precio actual de un producto en la base de datos de inventario. La entrada debe ser el SKU del producto."

¿Cómo se usa?

1. Creas una instancia de tu herramienta (puedes hacerlo en el diseñador de formularios si la registras como un componente, o por código).
2. En el TAIAgentsNode que quieres que use esta herramienta, en lugar de implementar el OnExecute, asignas tu instancia de la herramienta a la propiedad Tool del nodo.
- 3.

Cuando el grafo ejecute ese nodo, automáticamente llamará al método Execute de la herramienta asignada, pasándole el Input del nodo y esperando que la herramienta rellene el Output.

Con este último pilar, ahora tienes una visión completa de todas las piezas que conforman **MakerAI Agents**. Has aprendido sobre el orquestador, los nodos que realizan el trabajo, los enlaces que definen el flujo, la pizarra que comparte el estado y las herramientas que conectan tu grafo con el mundo real.

MÓDULO 3: ¡HOLA, MUNDO! - TU PRIMER GRAFO

En este módulo, dejaremos la teoría a un lado para construir nuestro primer grafo de IA funcional. El objetivo es simple, pero nos permitirá familiarizarnos con el flujo de trabajo básico de **MakerAI Agents**: colocar componentes, conectarlos y ejecutar el grafo.

Objetivo:

Crearemos un grafo simple que reciba un nombre como entrada (por ejemplo, "Mundo") y produzca un saludo personalizado como salida ("Hola, Mundo").

Componentes que usaremos:

- 1 x TAIAgents: El orquestador principal.
- 2 x TAIAgentsNode: Uno para el inicio (StartNode) y otro para el final (EndNode).
- 1 x TAIAgentsLink: Para conectar nuestros dos nodos.
- Componentes VCL/FMX estándar: TEdit para la entrada, TButton para ejecutar y TMemo para ver los registros de OnPrint.

Paso 1: Preparando el Escenario en el Formulario

1. Abre un nuevo proyecto VCL o FMX en Delphi.
2. Desde la Paleta de Componentes, en la pestaña "MakerAI", arrastra y suelta los siguientes componentes en tu formulario:
 - TAIAgents
 - TAIAgentsNode
 - TAIAgentsNode
 - TAIAgentsLink
3. Añade los componentes de interfaz de usuario: TEdit, TButton y TMemo.
4. **Renombra los componentes para mayor claridad.** Esto es una buena práctica que te ayudará enormemente en grafos más complejos. Selecciona cada componente y, en el Inspector de Objetos, cambia su propiedad Name:

```
TAIAgents1 -> AgentManager
TAIAgentsNode1 -> nodeStart
TAIAgentsNode2 -> nodeEnd
TAIAgentsLink1 -> linkStartToEnd
TMemo1 -> memoLog
TEdit1 -> editInput (y limpia su propiedad Text)
TButton1 -> btnRun (y cambia su Caption/Text a "Ejecutar")
```

Tu formulario debería verse más o menos así (los componentes no visuales aparecerán en la parte inferior o como iconos):

Paso 2: Configurando el Grafo

Ahora vamos a "cablear" nuestros componentes para definir la estructura del grafo. Todo esto se hace a través del Inspector de Objetos.

1. Configurar el Orquestador (AgentManager):

- Selecciona el componente AgentManager.
- Busca la propiedad StartNode. Haz clic en el desplegable y selecciona nodeStart.
- Busca la propiedad EndNode. Haz clic en el desplegable y selecciona nodeEnd.

2. Conectar los Nodos con el Enlace:

- Selecciona nodeStart.
- Busca su propiedad Next. En el desplegable, selecciona linkStartToEnd. Esto le dice al nodo inicial que, cuando termine, debe pasar el control al enlace.
- Selecciona linkStartToEnd.
- Busca su propiedad NextA. En el desplegable, selecciona nodeEnd. Esto le dice al enlace que su único destino es el nodo final.

¡Eso es todo! Acabas de definir visualmente un flujo nodeStart -> linkStartToEnd -> nodeEnd.

Paso 3: Implementando la Lógica

Ahora necesitamos decirle a nuestro nodeStart qué hacer. Aquí es donde escribiremos nuestro primer trozo de código de IA.

1. Implementar la lógica del nodeStart:

- Selecciona nodeStart en el formulario.
- Ve al Inspector de Objetos, a la pestaña de **Eventos**.
- Haz doble clic en el evento OnExecute. El IDE generará el esqueleto del procedimiento.
- Escribe el siguiente código:

```
procedure TForm1.nodeStartExecute(Node, BeforeNode: TAIAgentsNode;
  Link: TAIAgentsLink; Input: String; var Output: String);
begin
  // 1. Imprimimos un mensaje de log para ver que estamos aquí.
  //   Cualquier llamada a Print() en un nodo se redirige al evento OnPrint del
  //   orquestador.
  Node.Print(Format('Nodo Start: Recibido input "%s"', [Input]));

  // 2. La lógica principal: crear el saludo.
  //   El parámetro 'Input' contiene lo que se pasó al método Run().
  Output := 'Hola, ' + Input;

  // 3. Imprimimos el resultado que vamos a pasar al siguiente nodo.
  Node.Print(Format('Nodo Start: Generado output "%s"', [Output]));
end;
```

2. Conectar los Eventos del Orquestador:

Necesitamos decirle al AgentManager qué hacer con los mensajes de Print y qué hacer cuando el grafo termine.

- Selecciona AgentManager.
- En la pestaña de **Eventos**, haz doble clic en OnPrint. Escribe este código para mostrar los logs en nuestro TMemo:

```
procedure TForm1.AgentManagerPrint(Sender: TObject; Value: String);
begin
    // Mostramos el mensaje en el TMemo, asegurándonos de que sea seguro para hilos.
    TThread.Synchronize(nil,
        procedure
        begin
            memoLog.Lines.Add(Format('[%s] %s', [TObject(Sender).ClassName], Value));
        end);
end;
```

- Ahora, haz doble clic en el evento OnEnd. Aquí es donde recibiremos el resultado final de todo el proceso.

```
procedure TForm1.AgentManagerEnd(Node: TAIAgentsNode; Value: string);
begin
    // 'Node' es el último nodo que se ejecutó (en nuestro caso, nodeEnd).
    // 'Value' es el Output final de ese último nodo.
    TThread.Synchronize(nil,
        procedure
        begin
            ShowMessage('El grafo ha finalizado. Resultado final: ' + Value);
            btnRun.Enabled := True; // Reactivamos el botón
        end);
end;
```

- **Importante:** Nota el uso de TThread.Synchronize. Los eventos del agente se ejecutan en un hilo secundario. Cualquier interacción con la interfaz de usuario (UI) **debe** hacerse dentro de TThread.Synchronize para evitar errores de acceso concurrente.

Paso 4: ¡Ejecutar!

Lo único que nos queda es llamar al método Run del agente desde nuestro botón.

1. Haz doble clic en el btnRun en el diseñador de formularios.
2. Escribe el siguiente código en el evento OnClick:

```
procedure TForm1.btnRunClick(Sender: TObject);
begin
    memoLog.Clear;
    btnRun.Enabled := False; // Desactivamos el botón mientras se ejecuta.
    AgentManager.Run(editInput.Text);
end;
```

¡A Probarlo!

Ejecuta tu aplicación (F9).

1. Escribe "Mundo" en el TEdit.

2. Haz clic en el botón "Ejecutar".

Lo que verás:

1. El botón se desactivará.
2. En el TMemo, aparecerán los logs casi instantáneamente:

```
[TAIAgentsNode] Nodo Start: Recibido input "Mundo"  
[TAIAgentsNode] Nodo Start: Generado output "Hola, Mundo"
```

3. Aparecerá un ShowMessage con el texto: "El grafo ha finalizado. Resultado final: Hola, Mundo".
4. El botón "Ejecutar" se volverá a activar.

¡Felicidades! Acabas de construir y ejecutar tu primer Grafo de IA. Has aprendido a:

- Configurar el orquestador y definir el flujo del grafo.
- Implementar la lógica de un nodo.
- Manejar los eventos de log y finalización.
- Ejecutar el grafo de forma asíncrona.

Aunque este ejemplo es simple, los principios que has aplicado son la base para construir agentes y flujos de trabajo increíblemente complejos.

MÓDULO 4: TOMANDO DECISIONES - ENRUTAMIENTO CONDICIONAL

En el mundo real, los procesos rara vez son lineales. A menudo, necesitamos tomar un camino u otro basándonos en cierta información. En este módulo, construiremos un grafo que simula el análisis de sentimiento de un texto. Si el texto se considera "positivo", seguirá una ruta; si es "negativo", seguirá otra.

Objetivo:

Crear un grafo que reciba un texto. Un nodo central "analizará" si el texto contiene la palabra "bien" (simulando un análisis positivo) o "mal" (simulando uno negativo). Dependiendo del resultado, el flujo se dirigirá a un nodo de respuesta positiva o a uno de respuesta negativa.

Componentes que usaremos:

- 1 x TAIAgents (podemos reutilizar el del módulo anterior)
- 4 x TAIAgentsNode
- 3 x TAIAgentsLink

Paso 1: Diseño del Grafo en el Formulario

Vamos a modificar el formulario del módulo anterior o a crear uno nuevo.

1. Coloca y renombra los componentes:

```

AgentManager (TAIAgents)
nodeStart (TAIAgentsNode)
nodeAnalyze (TAIAgentsNode): Este será nuestro nodo de decisión.
nodePositiveResponse (TAIAgentsNode): Se ejecutará si el sentimiento es positivo.
nodeNegativeResponse (TAIAgentsNode): Se ejecutará si el sentimiento es negativo.
nodeEnd (TAIAgentsNode): Será el punto final común.
  
```

2. Configurar el Orquestador (AgentManager):

- StartNode: nodeStart
- EndNode: nodeEnd

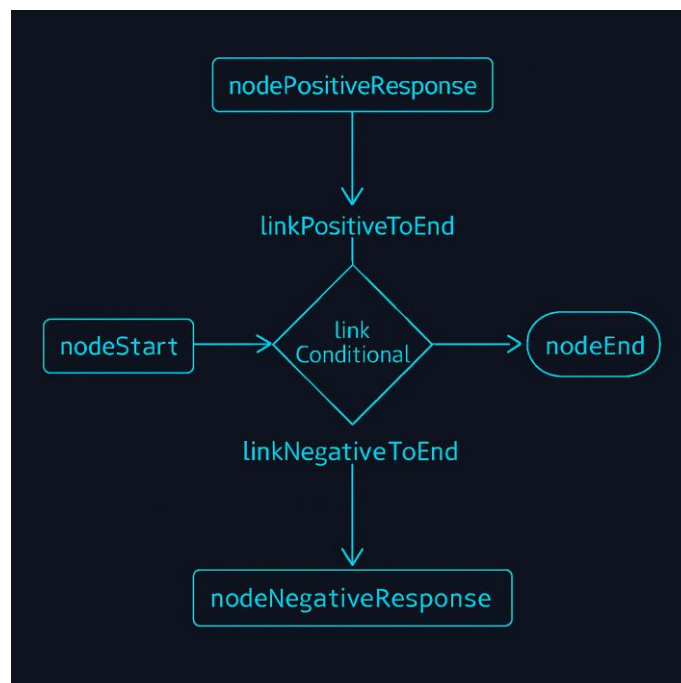
3. Cableado del Grafo (la parte crucial):

Este grafo tiene una bifurcación, así que presta atención a las conexiones.

- De nodeStart a nodeAnalyze:
 - Crea un enlace linkStartToAnalyze.
 - nodeStart.Next -> linkStartToAnalyze
 - linkStartToAnalyze.NextA -> nodeAnalyze
- La bifurcación desde nodeAnalyze:
 - Este es el paso más importante. nodeAnalyze tendrá un único enlace de salida, pero este enlace será **condicional**.
 - Crea un enlace y llámalo linkConditionalRouter.
 - nodeAnalyze.Next -> linkConditionalRouter

- ¡No configures NextA en linkConditionalRouter! Lo haremos por código para definir las rutas condicionales.
- De las ramas al nodeEnd:
 - Crea un enlace linkPositiveToEnd.
 - nodePositiveResponse.Next -> linkPositiveToEnd
 - linkPositiveToEnd.NextA -> nodeEnd
 - Crea un enlace linkNegativeToEnd.
 - nodeNegativeResponse.Next -> linkNegativeToEnd
 - linkNegativeToEnd.NextA -> nodeEnd

Visualmente, tu flujo se verá así:



Paso 2: Implementando la Lógica de Decisión

La magia del enrutamiento condicional reside en dos partes:

1. Un nodo escribe una "decisión" en el Blackboard.
2. Un enlace posterior lee esa decisión y enruta el flujo basándose en ella.
3. **Lógica del nodeStart:**

Este nodo simplemente pasará la entrada. Haz doble clic en su evento OnExecute.

```

procedure TForm1.nodeStartExecute(Node, BeforeNode: TAIAgentsNode;
  Link: TAIAgentsLink; Input: String; var Output: String);
begin
  Node.Print('Paso 1: Iniciando análisis con texto: ' + Input);
  // Simplemente pasamos la entrada al siguiente nodo
  Output := Input;
end;

```

4. **Lógica del nodeAnalyze (El Cerebro):**

Este es el nodo que toma la decisión.

```
procedure TForm1.nodeAnalyzeExecute(Node, BeforeNode: TAIAgentsNode;
  Link: TAIAgentsLink; Input: String; var Output: String);
var
  Decision: string;
begin
  // Simulamos un análisis de sentimiento
  if Pos('bien', Input.ToLower) > 0 then
    Decision := 'positivo'
  else if Pos('mal', Input.ToLower) > 0 then
    Decision := 'negativo'
  else
    Decision := 'neutro'; // Una ruta por defecto

  Node.Print('Paso 2: Análisis completado. Decisión: ' + Decision);

  // ¡Aquí está la clave! Escribimos la decisión en el Blackboard.
  // El enlace condicional usará esta clave para enrutar.
  Node.Graph.Blackboard.SetString('next_route', Decision);

  // Pasamos la decisión como salida para que los siguientes nodos la usen.
  Output := 'El sentimiento detectado fue: ' + Decision;
end;
```

- **Node.Graph.Blackboard:** Así es como accedemos al Blackboard desde un nodo.
- **'next_route':** Este es el nombre de clave que el sistema usa por defecto para el enrutamiento condicional.

5. Lógica de los Nodos de Respuesta:

Estos nodos simplemente generan un mensaje final basado en la ruta que se tomó.

```
procedure TForm1.nodePositiveResponseExecute(Node, BeforeNode: TAIAgentsNode;
  Link: TAIAgentsLink; Input: String; var Output: String);
begin
  Node.Print('Ruta Positiva: Generando respuesta optimista.');
```

```
  Output := '¡Gracias por tus comentarios positivos!';
end;
```

```
procedure TForm1.nodeNegativeResponseExecute(Node, BeforeNode: TAIAgentsNode;
  Link: TAIAgentsLink; Input: String; var Output: String);
begin
  Node.Print('Ruta Negativa: Generando respuesta de disculpa.');
```

```
  Output := 'Lamentamos que tu experiencia no haya sido buena.';
end;
```

Paso 3: Configurando el Enlace Condicional por Código

El diseñador de Delphi es genial para conexiones simples, pero las rutas condicionales se definen más fácilmente por código, normalmente en el evento OnCreate del formulario.

1. Ve al evento OnCreate de tu formulario.
2. Añade el siguiente código para "enseñarle" al linkConditionalRouter sus posibles destinos:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // Añadimos las rutas al enlace condicional
  linkConditionalRouter.AddConditionalTarget('positivo', nodePositiveResponse);
  linkConditionalRouter.AddConditionalTarget('negativo', nodeNegativeResponse);

  // Opcional: ¿Qué pasa si la decisión no es ni 'positivo' ni 'negativo'?
  // Podemos definir un destino por defecto. En este caso, si es 'neutro' o
  // cualquier otra cosa, irá directo al final.
```

```
linkConditionalRouter.NextNo := nodeEnd;
end;
```

- **AddConditionalTarget(Key, Node):** Este método le dice al enlace: "Cuando la clave 'next_route' del Blackboard contenga el valor Key, dirige el flujo al Node especificado".
- **NextNo:** Si el valor en el Blackboard no coincide con ninguna de las claves condicionales, el flujo tomará esta ruta alternativa. Si no la definimos y no hay coincidencia, el grafo se detendrá con un error.

Paso 4: ¡A Probarlo!

Usa la misma UI del módulo anterior (TEdit, TButton, TMemo) y el mismo código para los eventos OnPrint, OnEnd y OnClick del botón.

Prueba 1: Sentimiento Positivo

1. Escribe en el TEdit: "El servicio estuvo muy bien".
2. Haz clic en "Ejecutar".
3. **Log esperado:**

```
[TAIAgentsNode] Paso 1: Iniciando análisis...
[TAIAgentsNode] Paso 2: Análisis completado. Decisión: positivo
[TAIAgentsNode] Ruta Positiva: Generando respuesta optimista.
```

4. **Mensaje final:** "El grafo ha finalizado. Resultado final: ¡Gracias por tus comentarios positivos!"

Prueba 2: Sentimiento Negativo

1. Escribe en el TEdit: "Qué mal lo pasé".
2. Haz clic en "Ejecutar".
3. **Log esperado:**

```
[TAIAgentsNode] Paso 1: Iniciando análisis...
[TAIAgentsNode] Paso 2: Análisis completado. Decisión: negativo
[TAIAgentsNode] Ruta Negativa: Generando respuesta de disculpa.
```

4. **Mensaje final:** "El grafo ha finalizado. Resultado final: Lamentamos que tu experiencia no haya sido buena."

Prueba 3: Sentimiento Neutro

1. Escribe en el TEdit: "El día está nublado".
2. Haz clic en "Ejecutar".
3. **Log esperado:**

```
[TAIAgentsNode] Paso 1: Iniciando análisis...
[TAIAgentsNode] Paso 2: Análisis completado. Decisión: neutro
```

4. **Mensaje final:** "El grafo ha finalizado. Resultado final: El sentimiento detectado fue: neutro". (El flujo fue directo al nodeEnd a través de la ruta NextNo).

Acabas de construir un grafo que reacciona dinámicamente a sus datos de entrada. Este patrón es la base para crear agentes que pueden elegir qué herramienta usar, decidir si necesitan más información del usuario o manejar diferentes tipos de solicitudes de forma inteligente.

MÓDULO 5: DIVIDIR Y CONQUISTAR - EJECUCIÓN EN PARALELO (FAN-OUT)

A menudo, un problema requiere realizar varias tareas independientes que no dependen unas de otras. Por ejemplo, para crear un informe completo sobre un tema, podríamos querer generar un resumen, extraer las palabras clave y traducirlo a otro idioma. Ejecutar estas tareas una tras otra sería ineficiente.

MakerAI Agents simplifica enormemente la ejecución de tareas en paralelo (conocido como "Fan-out"). En este módulo, construiremos un grafo que toma un tema y lanza tres "trabajadores" simultáneamente para procesarlo de diferentes maneras.

Objetivo:

Crear un grafo que reciba un tema de entrada (ej. "Inteligencia Artificial"). Un enlace dividirá el flujo para que tres nodos se ejecuten al mismo tiempo:

1. Un nodo que simula la creación de un resumen.
2. Un nodo que simula la extracción de palabras clave.
3. Un nodo que simula la traducción del texto.

Importante: En este módulo nos centraremos solo en la parte de "dividir" (Fan-out). En el siguiente módulo, veremos cómo "unir" los resultados (Join).

Componentes que usaremos:

- 1 x TAIAgents
- 5 x TAIAgentsNode
- 4 x TAIAgentsLink

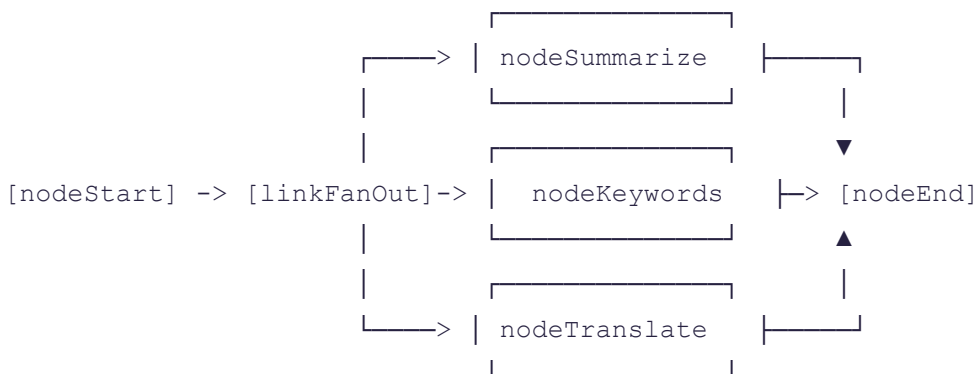
Paso 1: Diseño del Grafo Paralelo

1. **Coloca y renombra los componentes:**
 - AgentManager (TAIAgents)
 - nodeStart (TAIAgentsNode): Recibe el tema.
 - nodeSummarize (TAIAgentsNode): Tarea de resumen.
 - nodeKeywords (TAIAgentsNode): Tarea de palabras clave.
 - nodeTranslate (TAIAgentsNode): Tarea de traducción.
 - nodeEnd (TAIAgentsNode): Nuestro punto final.
2. **Configurar el Orquestador (AgentManager):**
 - StartNode: nodeStart
 - EndNode: nodeEnd
3. **Cableado del Grafo (Fan-out):**

- De nodeStart al enlace "divisor":
 - Crea un enlace linkFanOut.
 - nodeStart.Next -> linkFanOut
- ¡La Magia del Fan-out! Conectar linkFanOut a los tres nodos trabajadores:
 - Selecciona linkFanOut.
 - NextA -> nodeSummarize
 - NextB -> nodeKeywords
 - NextC -> nodeTranslate
- Conectar cada rama al final:
 - Para simplificar, por ahora haremos que cada rama termine de forma independiente (aunque no se unan, el grafo seguirá esperando a que todas terminen).
 - Crea linkSummarizeToEnd, linkKeywordsToEnd, linkTranslateToEnd.

```
nodeSummarize.Next -> linkSummarizeToEnd -> linkSummarizeToEnd.NextA -> nodeEnd
nodeKeywords.Next -> linkKeywordsToEnd -> linkKeywordsToEnd.NextA -> nodeEnd
nodeTranslate.Next -> linkTranslateToEnd -> linkTranslateToEnd.NextA -> nodeEnd
```

El flujo se verá así:



Nota: El nodeEnd en este diseño es un poco conceptual. Como tiene múltiples entradas, se ejecutará tan pronto como la primera rama llegue (por su JoinMode por defecto jmAny). El verdadero poder de este patrón se revelará en el próximo módulo cuando introduzcamos un nodo de unión (Join) antes del final.

Paso 2: Implementando la Lógica Paralela

Para demostrar que los nodos se ejecutan en paralelo, vamos a introducir un retardo artificial (Sleep) en cada uno. Si el flujo fuera secuencial, el tiempo total sería la suma de todos los retardos. En un flujo paralelo, el tiempo total será aproximadamente el del retardo más largo.

1. Lógica

del nodeStart:

Simplemente pasa el tema a linkFanOut.

```
procedure TForm1.nodeStartExecute(Node, BeforeNode: TAIAgentsNode;
  Link: TAIAgentsLink; Input: String; var Output: String);
begin
  Node.Print('Iniciando procesamiento del tema: ' + Input);
  Output := Input;
end;
```

2. Lógica de los Nodos Trabajadores:

Cada nodo realizará su "tarea", simulará trabajo con un Sleep y guardará su resultado en el Blackboard.

```
procedure TForm1.nodeSummarizeExecute(Node, BeforeNode: TAIAgentsNode;
  Link: TAIAgentsLink; Input: String; var Output: String);
begin
  Node.Print('>> [Hilo ' + TThread.Current.ThreadID.ToString + '] Iniciando resumen...');
  Sleep(2000); // Simula 2 segundos de trabajo
  Output := 'Este es un resumen sobre ' + Input + '.';
  Node.Graph.Blackboard.SetString('summary_result', Output);
  Node.Print('<< [Hilo ' + TThread.Current.ThreadID.ToString + '] Resumen finalizado.');
```

```
procedure TForm1.nodeKeywordsExecute(Node, BeforeNode: TAIAgentsNode;
  Link: TAIAgentsLink; Input: String; var Output: String);
begin
  Node.Print('>> [Hilo ' + TThread.Current.ThreadID.ToString + '] Iniciando extracción de palabras clave...');
  Sleep(3000); // Simula 3 segundos de trabajo
  Output := 'IA, Delphi, Paralelo';
  Node.Graph.Blackboard.SetString('keywords_result', Output);
  Node.Print('<< [Hilo ' + TThread.Current.ThreadID.ToString + '] Palabras clave finalizadas.');
```

```
procedure TForm1.nodeTranslateExecute(Node, BeforeNode: TAIAgentsNode;
  Link: TAIAgentsLink; Input: String; var Output: String);
begin
  Node.Print('>> [Hilo ' + TThread.Current.ThreadID.ToString + '] Iniciando traducción...');
  Sleep(1500); // Simula 1.5 segundos de trabajo
  Output := 'This is a text about ' + Input + '.';
  Node.Graph.Blackboard.SetString('translate_result', Output);
  Node.Print('<< [Hilo ' + TThread.Current.ThreadID.ToString + '] Traducción finalizada.');
```

- Hemos añadido el ThreadID al log. Si el paralelismo funciona, deberías ver diferentes IDs para cada nodo (o al menos, no siempre el mismo que el hilo principal).

Paso 3: ¡A Probarlo!

Usa la misma UI y los mismos eventos OnPrint, OnEnd y OnClick del botón de los módulos anteriores.

1. Escribe en el TEdit: "IA y Delphi".
2. Haz clic en "Ejecutar" y observa atentamente el memoLog.

Log Esperado:

Lo más probable es que veas los mensajes de "Iniciando..." aparecer casi todos a la vez. Luego, los mensajes de "finalizado" aparecerán en el orden en que terminen sus Sleep, no en el orden en que están definidos (Traducción -> Resumen -> Palabras Clave).

```
[TAIAgentsNode] Iniciando procesamiento del tema: IA y Delphi
[TAIAgentsNode] >> [Hilo 1234] Iniciando resumen...
[TAIAgentsNode] >> [Hilo 5678] Iniciando extracción de palabras clave...
[TAIAgentsNode] >> [Hilo 9012] Iniciando traducción...
// ... 1.5 segundos después ...
[TAIAgentsNode] << [Hilo 9012] Traducción finalizada.
// ... 0.5 segundos después (2s en total) ...
[TAIAgentsNode] << [Hilo 1234] Resumen finalizado.
// ... 1 segundo después (3s en total) ...
[TAIAgentsNode] << [Hilo 5678] Palabras clave finalizadas.
```

Observaciones Clave:

- **Tiempo de Ejecución:** El tiempo total de ejecución del grafo será de aproximadamente 3 segundos (la duración de la tarea más larga), no de 6.5 segundos (2 + 3 + 1.5). Esto demuestra la eficiencia ganada.
- **IDs de Hilo:** Notarás que los IDs de los hilos son diferentes, confirmando que TAIAgents ha utilizado el pool de hilos de TTask para ejecutar las ramas en paralelo.
- **OnEnd:** El evento OnEnd se disparará tan pronto como la primera rama (traducción) llegue al nodeEnd. El grafo principal, sin embargo, esperará internamente a que todas las tareas lanzadas terminen antes de considerarse completamente finalizado.

Has implementado con éxito un patrón de Fan-out, una técnica esencial para optimizar flujos de trabajo de IA que implican múltiples tareas independientes.

MÓDULO 6: JUNTANDO LAS PIEZAS - SINCRONIZACIÓN DE RAMAS (JOIN)

Este módulo es la contraparte natural del anterior. Después de haber dividido nuestro flujo en múltiples ramas paralelas (Fan-out), ahora aprenderemos a sincronizarlas en un único punto de unión (Join). Esto nos permite asegurarnos de que todas las tareas paralelas han finalizado antes de proceder con el siguiente paso, como, por ejemplo, compilar un informe final.

Objetivo:

Modificaremos el grafo del Módulo 5. En lugar de que cada rama vaya directamente al nodo final, las haremos converger en un nuevo "nodo de unión". Este nodo estará configurado para esperar a que las tres tareas (resumen, palabras clave, traducción) terminen. Una vez que todas hayan finalizado, se ejecutará para compilar un informe final a partir de los resultados almacenados en el Blackboard.

Concepto Clave: JoinMode = jmAll

La magia de la sincronización reside en la propiedad JoinMode del TAIAgentsNode.

- jmAny (por defecto): El nodo se ejecuta en cuanto llega la primera señal de entrada.
- jmAll: El nodo cuenta cuántas rutas de entrada tiene. Luego, espera pacientemente hasta haber recibido una señal de cada una de esas rutas antes de ejecutarse una sola vez.

Componentes que usaremos:

- Reutilizaremos la mayor parte del grafo del Módulo 5, pero con un nodo adicional y un re-cableado.
- 1 x TAIAgents
- 6 x TAIAgentsNode
- 5 x TAIAgentsLink (necesitamos menos enlaces porque ya no vamos directo al final desde cada rama)

Paso 1: Rediseñando el Grafo para la Unión

1. Añade el Nodo de Unión:

- Partiendo del diseño del Módulo 5, añade un nuevo TAIAgentsNode al formulario.
- Renómbralo a nodeReport.

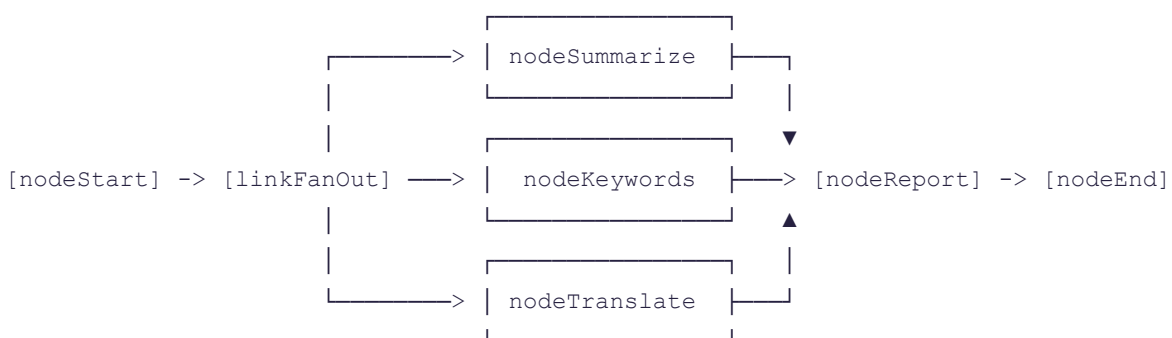
2. Configura el JoinMode:

- Selecciona nodeReport.
- En el Inspector de Objetos, busca la propiedad JoinMode y cámbiala de jmAny a jmAll. ¡Este es el paso más importante!

3. Re-cableado del Grafo:

- Vamos a redirigir la salida de los tres nodos trabajadores (nodeSummarize, nodeKeywords, nodeTranslate) para que apunten al nuevo nodeReport.
- **Desconecta de nodeEnd:** Elimina los enlaces linkSummarizeToEnd, linkKeywordsToEnd, y linkTranslateToEnd. O, más fácil, simplemente cambia sus propiedades.
- **Crea los nuevos enlaces hacia nodeReport:**
 - Crea un enlace linkSummarizeToReport.
 - nodeSummarize.Next -> linkSummarizeToReport
 - linkSummarizeToReport.NextA -> nodeReport
 - Crea un enlace linkKeywordsToReport.
 - nodeKeywords.Next -> linkKeywordsToReport
 - linkKeywordsToReport.NextA -> nodeReport
 - Crea un enlace linkTranslateToReport.
 - nodeTranslate.Next -> linkTranslateToReport
 - linkTranslateToReport.NextA -> nodeReport
- **Conecta el Nodo de Unión al Final:**
 - Crea un enlace linkReportToEnd.
 - nodeReport.Next -> linkReportToEnd
 - linkReportToEnd.NextA -> nodeEnd

El nuevo flujo se verá así:



Paso 2: Implementando la Lógica de Unión

La lógica de los nodos trabajadores (nodeSummarize, etc.) del módulo anterior ya está perfecta, ya que guardan sus resultados en el Blackboard. Ahora solo necesitamos implementar la lógica del nodeReport para que lea esos resultados y los combine.

1. Lógica del nodeReport:

- Selecciona nodeReport y haz doble clic en su evento OnExecute.

- Escribe el siguiente código:

```
procedure TForm1.nodeReportExecute(Node, BeforeNode: TAIAgentsNode;
  Link: TAIAgentsLink; Input: String; var Output: String);
var
  ReportBuilder: TStringBuilder;
  Summary, Keywords, Translation: string;
begin
  Node.Print('>>> ;Todas las ramas han finalizado! Compilando el informe final...');

  // Leemos los resultados que los nodos anteriores dejaron en el Blackboard.
  Summary := Node.Graph.Blackboard.GetString('summary_result', '[Resumen no disponible]');
  Keywords := Node.Graph.Blackboard.GetString('keywords_result', '[Palabras clave no
disponibles]');
  Translation := Node.Graph.Blackboard.GetString('translate_result', '[Traducción no
disponible]');

  // Construimos el informe final.
  ReportBuilder := TStringBuilder.Create;
  try
    ReportBuilder.AppendLine('--- INFORME COMPLETO ---');
    ReportBuilder.AppendLine('');
    ReportBuilder.AppendLine('Resumen:');
    ReportBuilder.AppendLine(Summary);
    ReportBuilder.AppendLine('');
    ReportBuilder.AppendLine('Palabras Clave:');
    ReportBuilder.AppendLine(Keywords);
    ReportBuilder.AppendLine('');
    ReportBuilder.AppendLine('Traducción (Inglés):');
    ReportBuilder.AppendLine(Translation);
    ReportBuilder.AppendLine('-----');

    // Asignamos el informe completo como la salida final del grafo.
    Output := ReportBuilder.ToString;
  finally
    ReportBuilder.Free;
  end;

  Node.Print('<<< Informe final compilado.');
```

Paso 3: ¡A Probarlo!

Usa la misma UI y configuración que en el módulo anterior. No necesitas cambiar nada más.

1. Escribe en el TEdit: "IA y Delphi".
2. Haz clic en "Ejecutar".

Log y Comportamiento Esperado:

1. Verás los mensajes de "Iniciando..." de los tres nodos trabajadores aparecer casi simultáneamente, igual que antes.
2. Verás los mensajes de "finalizado" de cada trabajador aparecer a medida que completan sus Sleeps (a los 1.5s, 2s y 3s).
3. **La diferencia clave:** Justo después de que el último trabajador (nodeKeywords) termine su ejecución (a los 3 segundos), verás aparecer inmediatamente los logs del nodeReport:

```
...
[TAIAgentsNode] << [Hilo 5678] Palabras clave finalizadas.
[TAIAgentsNode] >>> ;Todas las ramas han finalizado! Compilando el informe final...
[TAIAgentsNode] <<< Informe final compilado.
```

4. El ShowMessage del evento OnEnd ahora mostrará el informe completo y bien formateado que generó el nodeReport, en lugar de un resultado de una sola rama.

¡Éxito! Has implementado un patrón completo de Fan-out/Join. Has creado un grafo que:

1. Divide el trabajo en tareas paralelas para máxima eficiencia.
2. Espera de forma segura a que todas las tareas terminen.
3. Consolida los resultados de todas las ramas en una salida final unificada.

Este patrón es fundamental para aplicaciones de IA que necesitan recopilar y procesar información de múltiples fuentes antes de presentar una respuesta final al usuario.

MÓDULO 7: CREANDO AGENTES AUTÓNOMOS CON HERRAMIENTAS

Hasta ahora, toda nuestra lógica de negocio ha estado dentro de los eventos OnExecute de los nodos. Esto funciona para ejemplos sencillos, pero en aplicaciones reales, conduce a código duplicado y a una mezcla de responsabilidades (la lógica de orquestación mezclada con la lógica de negocio).

El componente TAIToolBase resuelve este problema. Nos permite encapsular una capacidad específica —como consultar una base de datos, llamar a una API web o leer un archivo— en un componente reutilizable. Luego, un nodo puede simplemente *usar* esa herramienta en lugar de implementar la lógica él mismo.

Objetivo:

Vamos a construir un agente simple que pueda realizar una tarea del "mundo real": consultar el tiempo.

1. Crearemos una nueva clase TWeatherTool que hereda de TAIToolBase.
2. Esta herramienta simulará una llamada a una API del tiempo.
3. Crearemos un grafo simple donde un nodo, en lugar de usar su evento OnExecute, utilizará la TWeatherTool para obtener la información.

Paso 1: Creando Nuestra Primera Herramienta (TWeatherTool)

Una herramienta es simplemente una clase Delphi. Podemos añadirla a una nueva unidad en nuestro proyecto para mantener el código organizado.

1. Añade una nueva unidad a tu proyecto (File -> New -> Unit). Guárdala como uMyTools.pas.
2. En esta nueva unidad, escribe el código para nuestra herramienta del tiempo.

```
// uMyTools.pas
unit uMyTools;

interface

uses
  System.SysUtils, System.Classes, uMakerAi.Agents; // Asegúrate de añadir uMakerAi.Agents

type
  TWeatherTool = class(TAIToolBase)
  protected
    // Este es el método que debemos implementar
    procedure Execute(ANode: TAIAgentsNode; const AInput: string; var AOutput: string);
  override;
  public
    constructor Create(AOwner: TComponent); override;
  end;

implementation

{ TWeatherTool }

constructor TWeatherTool.Create(AOwner: TComponent);
begin
```



```

inherited;
// Es una excelente práctica establecer la descripción aquí.
// Esta descripción es clave para que los LLMs puedan entender qué hace la herramienta.
Self.Description := 'Obtiene el pronóstico del tiempo para una ciudad específica. La
entrada debe ser el nombre de la ciudad.';
end;

procedure TWeatherTool.Execute(ANode: TAIAgentsNode; const AInput: string; var AOutput:
string);
var
    City: string;
begin
    City := AInput;
    ANode.Print(Format('TWeatherTool: Consultando tiempo para la ciudad "%s"...', [City]));

    // --- Simulación de llamada a una API del tiempo ---
    // En un caso real, aquí iría tu código con TRESTClient, etc.
    // Para este ejemplo, solo devolvemos datos fijos basados en la ciudad.
    Sleep(1500); // Simular latencia de red

    if City.ToLower.Contains('madrid') then
        AOutput := 'El tiempo en Madrid es soleado, 25°C.'
    else if City.ToLower.Contains('londres') then
        AOutput := 'El tiempo en Londres es nublado, 15°C.'
    else if City.ToLower.Contains('berlin') then
        AOutput := 'El tiempo en Berlín es lluvioso, 12°C.'
    else
        AOutput := Format('No se pudo encontrar el tiempo para la ciudad "%s".', [City]);

    ANode.Print('TWeatherTool: Consulta finalizada.');
```

Importante: Vuelve a tu unidad principal del formulario y añade uMyTools a la cláusula uses de la sección implementation.

Paso 2: Construyendo el Grafo que Usará la Herramienta

Ahora diseñaremos un grafo muy simple que utilice nuestra nueva herramienta.

1. Componentes en el Formulario:

- AgentManager (TAIAgents)
- nodeStart (TAIAgentsNode)
- nodeGetWeather (TAIAgentsNode)
- nodeEnd (TAIAgentsNode)
- linkStartToWeather (TAIAgentsLink)
- linkWeatherToEnd (TAIAgentsLink)

2. Añadir la Herramienta al Formulario:

- Para usar nuestra TWeatherTool, necesitamos una instancia de ella. Podemos crearla dinámicamente. En el evento OnCreate del formulario, añade:

```

// En la sección 'private' de la declaración de tu formulario
private
    FWeatherTool: TWeatherTool;

// En el evento OnCreate del formulario
procedure TForm1.FormCreate(Sender: TObject);
begin
    FWeatherTool := TWeatherTool.Create(Self); // Creamos la herramienta
    // ... (aquí puede ir otro código de inicialización)
```

```
end;

// No olvides liberarla en el OnDestroy
procedure TForm1.FormDestroy(Sender: TObject);
begin
    FWeatherTool.Free;
    inherited;
end;
```

(Alternativa para expertos: Podrías usar RegisterComponents para que tu herramienta aparezca en la paleta y puedas arrastrarla y soltarla como cualquier otro componente. Para este manual, la creación por código es más directa).

3. Cableado del Grafo:

```
AgentManager.StartNode -> nodeStart
AgentManager.EndNode -> nodeEnd
nodeStart.Next -> linkStartToWeather
linkStartToWeather.NextA -> nodeGetWeather
nodeGetWeather.Next -> linkWeatherToEnd
linkWeatherToEnd.NextA -> nodeEnd
```

4. Asignar la Herramienta al Nodo:

- Selecciona nodeGetWeather.
- Ve al Inspector de Objetos. Busca la propiedad Tool.
- **¡Este es el paso clave!** En lugar de usar el desplegable, vamos a asignarla por código para que sea explícito. En el FormCreate, después de crear la herramienta, añade:

```
nodeGetWeather.Tool := FWeatherTool;
```

Paso 3: Implementando la Lógica (Mínima)

La belleza de este enfoque es que casi no necesitamos escribir lógica en los eventos del formulario.

1. Lógica de nodeStart:

Este nodo solo pasa la entrada (el nombre de la ciudad).

```
procedure TForm1.nodeStartExecute(Node, BeforeNode: TAIAgentsNode;
    Link: TAIAgentsLink; Input: String; var Output: String);
begin
    Output := Input;
end;
```

2. Lógica de nodeGetWeather:

- **¡No implementes el evento OnExecute!**
- El nodo, al ejecutarse, verá que tiene una Tool asignada. Ignorará su (inexistente) OnExecute y en su lugar llamará al método Execute de nuestra TWeatherTool. El Input del nodo se pasará automáticamente al AInput de la herramienta, y el AOutput de la herramienta se convertirá en el Output del nodo.

Paso 4: ¡A Probarlo!

1. Usa la misma UI de los módulos anteriores.
2. Ejecuta la aplicación.
3. Escribe en el TEdit: "Quiero saber el tiempo en Madrid".
4. Haz clic en "Ejecutar".

Log y Resultado Esperado:

1. Log:

```
[TAIAgentsNode] TWeatherTool: Consultando tiempo para la ciudad "Quiero saber el tiempo en Madrid"...\n[TAIAgentsNode] TWeatherTool: Consulta finalizada.
```

2. **Mensaje Final:** "El grafo ha finalizado. Resultado final: El tiempo en Madrid es soleado, 25°C."

¡Análisis del Éxito!

Has separado con éxito la lógica de negocio de la orquestación.

- Tu grafo (TForm) ahora solo se preocupa del **flujo**: Start -> GetWeather -> End.
- La lógica compleja de **cómo** obtener el tiempo está encapsulada en TWeatherTool.

Ahora puedes reutilizar TWeatherTool en cualquier otro grafo. Si mañana la API del tiempo cambia, solo tienes que modificar la unidad uMyTools.pas, y todos tus grafos que la usen se actualizarán automáticamente sin necesidad de tocar la lógica del formulario.

Este patrón es la base para construir agentes complejos. Imagina tener un conjunto de herramientas: TDatabaseTool, TEmailTool, TCalendarTool. Puedes construir grafos que combinen estas herramientas para automatizar tareas sofisticadas, todo ello manteniendo tu código limpio, modular y escalable.

MÓDULO 8: GESTIÓN DE ERRORES Y CICLOS

Un sistema de IA robusto no es aquel que nunca falla, sino aquel que puede anticipar, manejar y recuperarse de los fallos de manera elegante. **MakerAI Agents** proporciona varios mecanismos para gestionar errores y controlar flujos repetitivos (ciclos) para evitar que se descontrolen.

Objetivo:

Construiremos un grafo que intenta realizar una "acción crítica" que puede fallar.

1. Si la acción tiene éxito, el grafo termina.
2. Si la acción falla, el flujo se redirigirá a un nodo de "manejo de errores" que registrará el problema.
3. Implementaremos un mecanismo de reintento controlado (ciclo) que intentará la acción un número máximo de veces antes de rendirse.

Conceptos Clave:

- **TAIAgents.OnError**: El evento global para capturar excepciones no controladas.
- **TAIAgentsLink.OnExecute**: El evento de un enlace donde podemos introducir lógica para determinar si una transición es exitosa (`IsOk := True`) o ha fallado (`IsOk := False`).
- **TAIAgentsLink.NextNo**: La ruta que toma el flujo cuando una transición falla (`IsOk` es `False`).
- **TAIAgentsLink.MaxCicles**: El guardián contra los bucles infinitos.

Paso 1: Diseño del Grafo Robusto

1. Componentes en el Formulario:

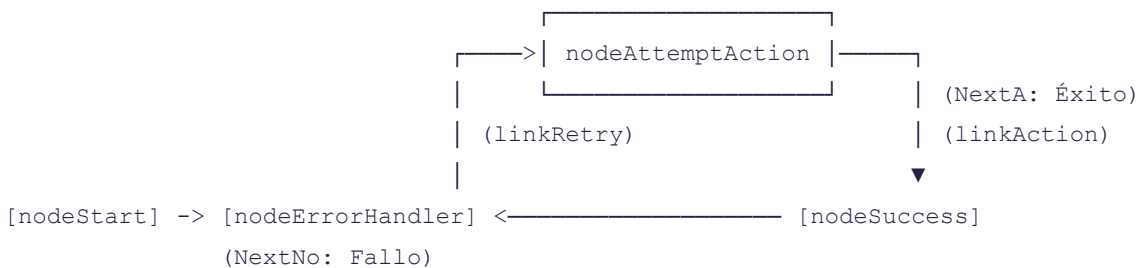
- AgentManager (TAIAgents)
- nodeStart (TAIAgentsNode)
- nodeAttemptAction (TAIAgentsNode): El nodo que intentará la acción crítica.
- nodeErrorHandler (TAIAgentsNode): El nodo al que iremos si la acción falla.
- nodeSuccess (TAIAgentsNode): El nodo final si todo va bien.

2. Cableado del Grafo (con un Ciclo):

- AgentManager.StartNode -> nodeStart
- AgentManager.EndNode -> nodeSuccess (¡Ojo! El final feliz es nodeSuccess).
- **De nodeStart a la acción:**
 - Crea linkStartToAction.
 - nodeStart.Next -> linkStartToAction

- linkStartToAction.NextA -> nodeAttemptAction
- **El enlace crítico (linkAction):** Este es el corazón de nuestro diseño.
 - Crea linkAction.
 - nodeAttemptAction.Next -> linkAction
 - **Ruta de Éxito:** linkAction.NextA -> nodeSuccess.
 - **Ruta de Fallo:** linkAction.NextNo -> nodeErrorHandler.
 - **Control de Ciclos:** Selecciona linkAction y en el Inspector de Objetos, establece MaxCycles a 3. Esto significa que permitiremos 3 intentos.
- **El Ciclo de Reintento:**
 - Para reintentar, necesitamos un enlace que vaya desde nuestro manejador de errores de vuelta al nodo de acción.
 - Crea linkRetry.
 - nodeErrorHandler.Next -> linkRetry
 - linkRetry.NextA -> nodeAttemptAction.

El flujo se verá así:



Paso 2: Implementando la Lógica de Fallo y Reintento

1. **Simulando un fallo:** Para probar esto, necesitamos una forma de hacer que nuestra "acción crítica" falle de vez en cuando. Usaremos un contador simple en nuestro formulario.
 - Añade una variable privada al tu TForm: FAttemptCounter: Integer;
 - En el OnClick del botón btnRun, resetea el contador: FAttemptCounter := 0;
2. **Lógica del nodeAttemptAction:**

Este nodo simplemente registra que está intentando la acción.

```

procedure TForm1.nodeAttemptActionExecute(Node, BeforeNode: TAIAgentsNode;
  Link: TAIAgentsLink; Input: String; var Output: String);
begin
  Inc(FAttemptCounter);
  Node.Print(Format('Intento #%d: Realizando acción crítica...', [FAttemptCounter]));
  // El resultado de este nodo no es importante, la decisión se toma en el enlace.
  Output := 'Acción intentada';
end;

```

3. **Lógica del Enlace Crítico (linkAction.OnExecute):**

Aquí es donde decidimos si el intento fue exitoso o no.

```

procedure TForm1.linkActionExecute(Node: TAIAgentsNode; Link: TAIAgentsLink;
  var IsOk: Boolean; var Handled: Boolean);
begin
  // Simulamos que la acción solo tiene éxito al tercer intento.
  if FAttemptCounter >= 3 then
  begin
    Node.Print('¡ÉXITO! La acción crítica funcionó.');
```

```

    IsOk := True; // ;El flujo continuará por NextA hacia nodeSuccess!
  end
  else
  begin
    Node.Print('FALLO. La acción crítica no funcionó esta vez.');
```

4. Lógica del nodeErrorHandler:

Este nodo se ejecuta cuando IsOk es False. Su trabajo es registrar el error y luego pasar el control al enlace de reintento.

```

procedure TForm1.nodeErrorHandlerExecute(Node, BeforeNode: TAIAgentsNode;
  Link: TAIAgentsLink; Input: String; var Output: String);
begin
  Node.Print('Manejador de Errores: La acción falló. Preparando reintento...');
```

```

  Sleep(1000); // Esperar un segundo antes de reintentar.
end;
```

5. Lógica del nodeSuccess:

Este es nuestro final feliz.

```

procedure TForm1.nodeSuccessExecute(Node, BeforeNode: TAIAgentsNode;
  Link: TAIAgentsLink; Input: String; var Output: String);
begin
  Output := Format('Proceso completado con éxito en el intento #%d.', [FAttemptCounter]);
end;
```

Paso 3: Manejando Excepciones Fatales (OnError)

¿Qué pasa si ocurre un error que no controlamos, como una división por cero o un objeto nulo? O, ¿qué pasa si excedemos MaxCicles? Para eso está el evento OnError del AgentManager.

1. Selecciona AgentManager y haz doble clic en su evento OnError.

```

procedure TForm1.AgentManagerError(Sender: TObject; Node: TAIAgentsNode;
  Link: TAIAgentsLink; E: Exception; var Abort: Boolean);
begin
  var ErrorMsg := '¡ERROR FATAL! ' + E.Message;
  if Assigned(Node) then
    ErrorMsg := ErrorMsg + ' en Nodo: ' + Node.Name
  else if Assigned(Link) then
    ErrorMsg := ErrorMsg + ' en Enlace: ' + Link.Name;

  TThread.Synchronize(nil, procedure
  begin
    memoLog.Lines.Add(ErrorMsg);
    ShowMessage('El grafo ha fallado de forma irre recuperable.');
```

```

    btnRun.Enabled := True;
  end);
```

```

  Abort := True; // Le decimos al agente que detenga toda la ejecución.
```

```
end;
```

Paso 4: ¡A Probarlo!

1. Ejecuta la aplicación. No necesitas escribir nada en el TEdit.
2. Haz clic en "Ejecutar".

Log Esperado:

Verás el ciclo en acción. El grafo intentará, fallará, irá al manejador de errores, reintentará, y así sucesivamente, hasta que finalmente tenga éxito.

```
[TAIAgentsNode] Intento #1: Realizando acción crítica...
[TAIAgentsNode] FALLO. La acción crítica no funcionó esta vez.
[TAIAgentsNode] Manejador de Errores: La acción falló. Preparando reintento...
[TAIAgentsNode] Intento #2: Realizando acción crítica...
[TAIAgentsNode] FALLO. La acción crítica no funcionó esta vez.
[TAIAgentsNode] Manejador de Errores: La acción falló. Preparando reintento...
[TAIAgentsNode] Intento #3: Realizando acción crítica...
[TAIAgentsNode] ¡ÉXITO! La acción crítica funcionó.
Mensaje Final: "El grafo ha finalizado. Resultado final: Proceso completado con éxito en el
intento #3."
```

Prueba de MaxCicles:

Si modificas la lógica de `linkActionExecute` para que *siempre* falle (`isOk := False;`), al ejecutar verás dos reintentos. En el tercer fallo, en lugar de continuar, el enlace `linkAction` detectará que se ha alcanzado `MaxCicles`. Lanzará una excepción interna que será capturada por tu evento `AgentManager.OnError`, y verás el mensaje de error fatal. Has aprendido a construir grafos que no solo ejecutan tareas, sino que también son resistentes a fallos, capaces de reintentar y de manejar errores inesperados de forma controlada.

APÉNDICE A: REFERENCIA DE LA API

Este apéndice sirve como una guía de referencia técnica para los principales componentes, propiedades, métodos y eventos del framework **MakerAI Agents**.

A.1 TAIAgents (El Orquestador)

Componente principal que gestiona la ejecución y el estado de un grafo de IA.

Propiedades Publicadas:

- **StartNode:** TAIAgentsNode: **(Requerido)** El primer nodo que se ejecutará al llamar a Run.
- **EndNode:** TAIAgentsNode: **(Requerido)** El nodo que, al ser alcanzado, marca una finalización nominal del grafo. Su OnExecute no se ejecuta automáticamente, pero el evento OnEnd del orquestador sí se dispara.
- **OnPrint:** TAIAgentsOnPrint: Evento para logging. Se dispara cuando cualquier nodo o enlace llama a su método Print.
- **OnEnd:** TAIAgentsOnEnd: Evento que se dispara cuando el grafo completa su ejecución. Recibe el último nodo ejecutado y su Output.
- **OnError:** TAIAgentsOnError: Evento global para manejar excepciones no controladas dentro de la ejecución del grafo.
- **OnConfirm:** TAIAgentsOnConfirm: Evento para facilitar la interacción con la UI. Permite pausar el grafo y solicitar confirmación o datos del usuario.

Propiedades de Solo Lectura:

- **Busy:** Boolean: Devuelve True si el grafo está actualmente en ejecución.
- **Blackboard:** TAIBlackboard: Proporciona acceso a la instancia de la pizarra compartida para esta ejecución del grafo.

Métodos Públicos:

- **Run(Msg: String): ITask:** Inicia la ejecución del grafo de forma asíncrona.
 - **Msg:** El string inicial que se pasará como Input al StartNode.
 - **Devuelve:** Una ITask que representa la ejecución en segundo plano. Puedes usar TTask.WaitForAll si necesitas esperar a que termine.
- **Abort:** Solicita la detención del grafo en ejecución. La detención no es inmediata, pero los nodos y enlaces dejarán de ejecutarse en el siguiente punto de control.
- **Compile:** Analiza la estructura del grafo, valida las conexiones y prepara las dependencias internas. Se llama automáticamente dentro de Run si el grafo no ha sido compilado.

- `FindNode(const AName: string): TAIAgentsNode`: Busca y devuelve un nodo por su nombre. Útil para la construcción de grafos por código.
 - `ClearGraph`: Elimina todos los nodos y enlaces creados dinámicamente.
-

A.2 TAIAgentsNode (El Nodo)

Representa una unidad de trabajo o un paso en el grafo.

Propiedades Publicadas:

- `Next: TAIAgentsLink`: El enlace de salida que se ejecutará después de este nodo.
- `Input: String`: Los datos de entrada para el nodo. Normalmente se rellena con el Output del nodo anterior.
- `Output: String`: El resultado del trabajo del nodo. Se pasará como Input al siguiente.
- `OnExecute: TAIAgentsNodeOnExecute`: El evento principal donde se define la lógica del nodo. **Tiene prioridad sobre la propiedad Tool.**
- `Tool: TAIToolBase`: Asigna un componente de herramienta reutilizable para que ejecute la lógica del nodo. Solo se usa si `OnExecute` no está asignado.
- `JoinMode: TJoinMode`: (`jmAny`, `jmAll`) Define el comportamiento del nodo cuando tiene múltiples enlaces de entrada. `jmAll` hace que el nodo espere a que todas las ramas lleguen antes de ejecutarse.
- `PromptName: String`: Propiedad de propósito general para asociar un nombre (por ejemplo, de un prompt) al nodo. No tiene funcionalidad interna.

Métodos Públicos:

- `Print(Value: String)`: Envía un mensaje de log al evento `OnPrint` del orquestador.
 - `RequestConfirmation(...)`: Solicita interacción al usuario a través del evento `OnConfirm` del orquestador.
 - `RequestInput(...)`: Un ayudante sobre `RequestConfirmation` para pedir un valor de texto al usuario.
 - `ForceFinalExecute`: Fuerza la ejecución del evento `OnEnd` del orquestador, usando este nodo como el finalizador. Útil para terminaciones abruptas.
-

A.3 TAIAgentsLink (El Enlace)

Define el flujo y la lógica de transición entre nodos.

Propiedades Publicadas:

- `NextA: TAIAgentsNode`: El destino principal o el primer destino en una ejecución paralela (Fan-out).

- NextB, NextC, NextD: TAIAgentsNode: Destinos adicionales para la ejecución en paralelo.
- NextNo: TAIAgentsNode: El destino que se tomará si la lógica del enlace determina un fallo (ej. `IsOk := False` en `OnExecute` o se supera `MaxCicles`).
- MaxCicles: Integer (Default: 1): El número máximo de veces que el flujo puede pasar a través de este enlace en un ciclo antes de lanzar un error de "máximos ciclos alcanzados". Un valor de 1 efectivamente deshabilita los ciclos a través de este enlace.
- OnExecute: TAIAgentsLinkOnExecute: Evento para implementar lógica de transición personalizada. Permite decidir si la transición es válida (`IsOk`) o si se debe detener el flujo (`Handled`).

Métodos Públicos:

- `AddConditionalTarget(const AKey: string; ANode: TAIAgentsNode)`: Añade una ruta al diccionario de enrutamiento condicional. El enlace tomará esta ruta si el valor de la clave 'next_route' en el Blackboard coincide con AKey.
- `Print(Value: String)`: Envía un mensaje de log al evento `OnPrint` del orquestador.

A.4 TAIBlackboard (La Pizarra)

Clase de acceso al estado compartido del grafo. Accesible a través de `TAIAgents.Blackboard`.

Métodos Públicos Principales:

- `SetString(const AKey, AValue: string)`
- `GetString(const AKey: string; const ADefault: string = ""): string`
- `SetInteger(const AKey: string; AValue: Integer)`
- `GetInteger(const AKey: string; const ADefault: Integer = 0): Integer`
- `SetBoolean(const AKey: string; AValue: Boolean)`
- `GetBoolean(const AKey: string; const ADefault: Boolean = False): Boolean`
- `SetValue(const AKey: string; const AValue: TValue)`: El método genérico para guardar cualquier tipo compatible con TValue.
- `TryGetValue(const AKey: string; out AValue: TValue): Boolean`: El método genérico para leer de forma segura cualquier tipo.
- `Clear`: Vacía todo el contenido del Blackboard. Se llama automáticamente al inicio de `Run`.

APÉNDICE B: CREACIÓN DE HERRAMIENTAS PERSONALIZADAS (TAITOOLBASE)

Mientras que el evento OnExecute de un nodo es perfecto para lógica única y específica del formulario, el verdadero poder de la modularidad y la reutilización se desbloquea mediante la creación de **Herramientas** personalizadas. Una herramienta bien diseñada encapsula una funcionalidad de negocio o técnica, ocultando su complejidad interna y proporcionando una interfaz simple al grafo.

Este apéndice cubre las mejores prácticas y técnicas avanzadas para construir tus propias herramientas.

B.1 La Estructura Fundamental de una Herramienta

Como vimos en el Módulo 7, la estructura básica de una herramienta es increíblemente simple. Heredas de TAIToolBase e implementas el método Execute.

```
unit uMyDataAccessTools;

interface

uses
  System.SysUtils, System.Classes, Data.DB, uMakerAi.Agents;

type
  TDatabaseQueryTool = class(TAIToolBase)
  private
    FConnection: TFDConnection; // Ejemplo con FireDAC
    FQuery: TFDQuery;
  protected
    procedure Execute(ANode: TAIAgentsNode; const AInput: string; var AOutput: string);
  override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    // Propiedades para configurar la herramienta en el Inspector de Objetos
    property Connection: TFDConnection read FConnection write FConnection;
  end;

implementation
// ... implementación ...
end;
```

Mejores Prácticas en el Diseño:

1. **Principio de Responsabilidad Única (SRP):** Una herramienta debe hacer **una sola cosa**, y hacerla bien.
 - o **Mal:** TDoEverythingTool que consulta la DB, llama a una API y escribe un archivo.

- **Bien:** Tres herramientas separadas: TDatabaseQueryTool, TWebApiGetTool, TFileWriterTool. Esto las hace más fáciles de probar, depurar y componer en diferentes grafos.
- 2. **Configuración a través de Propiedades:** Expón cualquier parámetro de configuración (como objetos de conexión, URLs de API, nombres de archivo, etc.) como propiedades published. Esto permite configurar la herramienta fácilmente desde el Inspector de Objetos de Delphi si la registras como un componente.
- 3. **La Descripción es Clave:** La propiedad Description es vital. No es un simple comentario. Es la **semántica** de la herramienta. Escribe una descripción clara y concisa que explique:
 - Qué hace la herramienta.
 - Qué espera exactamente como Input.
 - Qué devuelve como Output.
 - *Ejemplo:* ``Ejecuta una consulta SQL de solo lectura (SELECT). El Input debe ser la sentencia SQL completa. Devuelve el resultado de la primera columna de la primera fila, o una cadena vacía si no hay resultados."*

B.2 Interactuando con el Grafo desde una Herramienta

El método Execute recibe ANode: TAIAgentsNode como parámetro. Este objeto es tu puerta de enlace para interactuar con el resto del grafo.

- **Logging:** Usa ANode.Print('...') para enviar mensajes al log del orquestador. Esto es crucial para la depurabilidad. No uses ShowMessage o Writeln dentro de una herramienta, ya que se ejecuta en un hilo secundario y no tiene acceso a la consola o UI.
- **Acceso al Blackboard:** Puedes leer y escribir en el Blackboard a través de ANode.Graph.Blackboard. Esto es extremadamente útil para que una herramienta devuelva datos complejos o múltiples valores.

Ejemplo Avanzado: Herramienta que devuelve múltiples valores

```
procedure TUserInfoTool.Execute(ANode: TAIAgentsNode; const AInput: string; var AOutput:
string);
var
  UserID: Integer;
  UserName, UserEmail: string;
begin
  try
    UserID := StrToInt(AInput);
  except
    AOutput := 'Error: El input debe ser un ID de usuario numérico.';
    Exit;
  end;

  ANode.Print(Format('Buscando información para el usuario ID: %d', [UserID]));
```

```
// --- Lógica para buscar el usuario en la base de datos ---
// (Simulado para el ejemplo)
FindUserInDB(UserID, UserName, UserEmail);

// Escribimos los datos complejos en el Blackboard para que otros nodos los usen.
ANode.Graph.Blackboard.SetString('current_user_name', UserName);
ANode.Graph.Blackboard.SetString('current_user_email', UserEmail);

// El Output principal puede ser un resumen simple o un estado.
AOutput := Format('Información del usuario %s cargada con éxito.', [UserName]);
end;
En este ejemplo, la herramienta realiza su trabajo y luego "enriquece" el estado del grafo
(el Blackboard) con información detallada que puede ser utilizada por nodos posteriores en
el flujo.
```

B.3 Registrando tus Herramientas como Componentes

Para una integración perfecta con el IDE de Delphi, puedes registrar tus herramientas para que aparezcan en la Paleta de Componentes. Esto te permite arrastrarlas a tu formulario o DataModule, configurarlas visualmente y asignarlas a los nodos a través del desplegable del Inspector de Objetos.

1. Crea una unidad de registro (por ejemplo, uMyToolsRegister.pas).
2. Añade el siguiente código:

```
unit uMyToolsRegister;

interface

procedure Register;

implementation

uses System.Classes, uMyDataAccessTools, uMyApiTools; // Unidades donde están tus
herramientas

procedure Register;
begin
    // Registra tus herramientas en una pestaña de la Paleta de Componentes.
    // Puedes usar una pestaña existente o crear una nueva.
    RegisterComponents('MakerAI Tools', [TDatabaseQueryTool, TWebApiGetTool]);
end;

end.
```

Añade esta unidad de registro a tu paquete de diseño (.dpk) o instálala en el IDE. Una vez hecho, tus herramientas estarán disponibles como cualquier otro componente visual o no visual.

APÉNDICE C: CONSTRUCCIÓN DE GRAFOS DINÁMICAMENTE (FLUENT API)

Si bien el diseñador de formularios de Delphi es una herramienta fantástica para definir grafos estáticos, hay escenarios en los que necesitas construir, modificar o ejecutar un grafo completamente por código. **MakerAI Agents** expone una API fluida (fluent API) que te permite encadenar llamadas a métodos para definir la estructura de un grafo de una manera legible y concisa.

Esto es ideal para:

- Crear flujos de trabajo basados en reglas de negocio cargadas desde una base de datos.
 - Generar grafos dinámicamente en respuesta a la entrada del usuario.
 - Integrar la orquestación en sistemas donde no hay una interfaz de usuario visual (como servicios de Windows o servidores web).
 - Realizar pruebas unitarias de la lógica del grafo.
-

C.1 Los Bloques de Construcción de la API Fluida

El componente **TAIAgents** tiene una serie de métodos que, al devolver **Self**, te permiten encadenar la siguiente llamada.

- `AddNode(const AName: string; AExecuteProc: TAIAgentsNodeOnExecute): TAIAgents;`
 - Crea un nuevo **TAIAgentsNode** y lo añade al grafo.
 - **AName**: Un nombre único para el nodo. Es crucial, ya que lo usarás para referenciarlo al crear los enlaces.
 - **AExecuteProc**: Un puntero al procedimiento o método anónimo que implementará la lógica **OnExecute** del nodo.
- `AddEdge(const AStartNodeName, AEndNodeName: string): TAIAgents;`
 - Crea un **TAIAgentsLink** simple que conecta dos nodos por sus nombres.
- `AddConditionalEdge(const AStartNodeName: string; const AConditionalLinkName: string; AConditionalTargets: TDictionary<string, string>): TAIAgents;`
 - Crea un enlace condicional.
 - **AStartNodeName**: El nodo de origen.
 - **AConditionalLinkName**: Un nombre para el nuevo enlace.
 - **AConditionalTargets**: Un diccionario donde la clave es el valor de decisión (que se leerá de `Blackboard['next_route']`) y el valor es el nombre del nodo de destino.

- SetEntryPoint(const ANodeName: string): TAIAgents;
 - Define cuál de los nodos creados será el StartNode del grafo.
- SetFinishPoint(const ANodeName: string): TAIAgents;
 - Define cuál será el EndNode.

C.2 Ejemplo Práctico: Recreando el Grafo Condicional por Código

Vamos a recrear el grafo de análisis de sentimiento del Módulo 4, pero esta vez sin usar ningún componente en tiempo de diseño, excepto un TAIAgents (AgentManager), un TButton y un TMemo.

```

procedure TForm1.btnRunDynamicGraphClick(Sender: TObject);
begin
    memoLog.Clear;

    // 1. Limpiamos cualquier grafo definido previamente
    AgentManager.ClearGraph;

    // 2. Definimos la lógica de nuestros nodos usando métodos anónimos
    var AnalyzeLogic: TAIAgentsNodeOnExecute :=
        procedure (Node, BeforeNode: TAIAgentsNode; Link: TAIAgentsLink; Input: String; var
        Output: String)
        var
            Decision: string;
        begin
            if Pos('bien', Input.ToLower) > 0 then Decision := 'positivo'
            else if Pos('mal', Input.ToLower) > 0 then Decision := 'negativo'
            else Decision := 'neutro';
            Node.Print('Nodo de Análisis (dinámico): Decisión -> ' + Decision);
            Node.Graph.Blackboard.SetString('next_route', Decision);
            Output := Input;
        end;

    var PositiveLogic: TAIAgentsNodeOnExecute :=
        procedure (Node, BeforeNode: TAIAgentsNode; Link: TAIAgentsLink; Input: String; var
        Output: String)
        begin
            Output := 'Respuesta positiva generada dinámicamente.';
        end;

    var NegativeLogic: TAIAgentsNodeOnExecute :=
        procedure (Node, BeforeNode: TAIAgentsNode; Link: TAIAgentsLink; Input: String; var
        Output: String)
        begin
            Output := 'Respuesta negativa generada dinámicamente.';
        end;

    // 3. Construimos el grafo usando la API fluida
    AgentManager.AddNode('Start', nil) // Un nodo de inicio simple sin lógica
        .AddNode('Analyze', AnalyzeLogic)
        .AddNode('PositiveResponse', PositiveLogic)
        .AddNode('NegativeResponse', NegativeLogic)
        .AddNode('End', nil) // Un nodo final simple
        .AddEdge('Start', 'Analyze') // Conectamos el inicio con el análisis
        .AddConditionalEdge('Analyze', 'RouterLink',
            TDictionary<string, stringvar RouterLink := TAIAgentsLink(AgentManager.FindComponent('RouterLink'));
    if Assigned(RouterLink) then
        RouterLink.NextNo := AgentManager.FindNode('End');

```

```
// 4. Compilamos y ejecutamos el grafo recién creado
AgentManager.Compile;
AgentManager.Run('el servicio estuvo muy bien');
end;
```

Análisis del Código:

1. **Lógica Desacoplada:** Definimos la funcionalidad de cada nodo como variables de tipo `TAIAgentsNodeOnExecute`. Esto mantiene la lógica separada de la estructura. Podrían ser métodos de una clase en lugar de procedimientos anónimos para una mejor organización.
2. **Construcción Encadenada:** Observa cómo cada llamada (`.AddNode`, `.AddEdge`, etc.) se encadena a la anterior, haciendo que la definición de la estructura del grafo sea muy legible y se asemeje a una "receta".
3. **Flexibilidad Total:** Podrías cargar los nombres de los nodos, sus conexiones y la lógica a ejecutar desde un archivo JSON, una base de datos o cualquier otra fuente de configuración externa, permitiendo la creación de sistemas de orquestación de IA completamente dinámicos.

Este enfoque es, sin duda, más avanzado, pero desbloquea el nivel más alto de flexibilidad y poder de la librería **MakerAI Agents**, permitiéndote integrarla en arquitecturas de software complejas y dinámicas.

GLOSARIO DE TÉRMINOS

Esta sección proporciona definiciones concisas de los términos y componentes clave utilizados a lo largo del manual de **MakerAI Agents**.

A

- **Agente** (Agent)
Un sistema o programa que utiliza un Modelo de Lenguaje Grande (LLM) como su motor de razonamiento para percibir su entorno, tomar decisiones, y ejecutar acciones a través de Herramientas para alcanzar un objetivo.
- **API Fluida** (Fluent API)
Un estilo de diseño de interfaz de programación en el que la legibilidad del código fuente es cercana a la de la prosa ordinaria. Se logra encadenando llamadas a métodos (objeto.Metodo1().Metodo2().Metodo3()). En MakerAI Agents, se utiliza para la construcción de grafos por código.

B

- **Blackboard** (Pizarra)
Un repositorio de datos centralizado y compartido, específico para una única ejecución del grafo. Actúa como la "memoria a corto plazo" del agente, permitiendo que los Nodos y Enlaces se comuniquen entre sí, pasen datos complejos y mantengan el estado. Corresponde al componente TAIBlackboard.

E

- **Enlace** (Link / Edge)
El componente que conecta dos o más Nodos, definiendo la dirección del flujo. Un enlace puede contener lógica para enrutamiento condicional, ejecución en paralelo (Fan-out) y manejo de fallos. Corresponde al componente TAIagentsLink.
- **Enrutamiento** Condicional
La capacidad de un Enlace para dirigir el flujo a diferentes Nodos de destino basándose en una condición, típicamente un valor almacenado en el Blackboard. Es el mecanismo principal para la toma de decisiones en un grafo.

F

- **Fan-out** (Ejecución en Paralelo)
El proceso de dividir el flujo de un grafo en múltiples ramas que se ejecutan simultáneamente. Se logra configurando un único Enlace con múltiples Nodos de destino (NextA, NextB, etc.).

G

- **Grafo** (Graph)
La estructura completa que define un flujo de trabajo, compuesta por Nodos (los pasos) y Enlaces (las conexiones). A diferencia de una cadena lineal, un grafo puede tener ramas, ciclos y uniones, permitiendo modelar procesos complejos. El grafo es gestionado por el componente TAIAgents.

H

- **Herramienta** (Tool)
Un componente reutilizable que encapsula una capacidad específica para interactuar con sistemas externos (bases de datos, APIs, archivos, etc.). Las herramientas promueven la reutilización de código y la separación de responsabilidades. Corresponde a la clase base TAIToolBase.

J

- **Join** (Unión / Sincronización)
El proceso de esperar a que múltiples ramas paralelas (creadas por un Fan-out) completen su ejecución antes de continuar con un único paso siguiente. Se logra configurando un Nodo con la propiedad JoinMode en jmAll.

L

- **LLM** (Large Language Model / Modelo de Lenguaje Grande)
Un modelo de inteligencia artificial entrenado con enormes cantidades de texto, capaz de comprender y generar lenguaje humano. Es el "cerebro" o motor de razonamiento que un Agente utiliza para tomar decisiones. Ejemplos: GPT-4, Gemini, Llama.

N

- **Nodo** (Node)
La unidad fundamental de trabajo en un grafo. Representa un paso o una acción específica en el flujo de trabajo, como procesar datos, llamar a un LLM o ejecutar una Herramienta. Corresponde al componente TAIAgentsNode.

O

- **Orquestación** (Orchestration)
El proceso de coordinar y gestionar una secuencia de tareas complejas y distribuidas. En el contexto de la IA, se refiere a la gestión del flujo de llamadas a LLMs, herramientas y otras lógicas para resolver un problema que va más allá de una simple pregunta y respuesta. MakerAI Agents es un framework de orquestación.

S

- **Estado** (State)
La información acumulada durante la ejecución de un grafo. Se gestiona a través del

Blackboard y permite que el grafo "recuerde" resultados, decisiones y datos a medida que avanza por los diferentes Nodos.