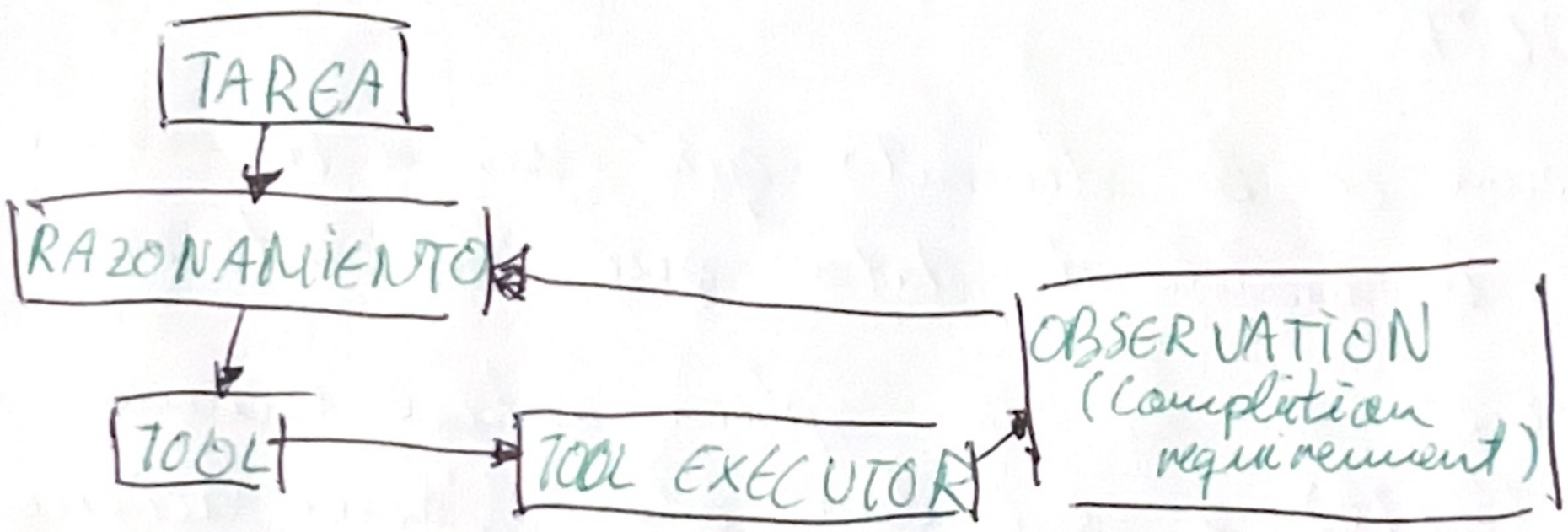
AGENTS LANGGRAPH

Un agente consta de 3 partes:

- LLM
- TOOLS
- PROMPT

Los agentes en LangGraph son básicamente un LOOP siguiendo una serie de pasos hasta completar la tarea.



Tenemos 2 tipos de memoria

- Short Term
- Long Term

- Human in the loop significa la intervención del humano en el loop workflow.
- Streamin Support (Es para monitorear lo que hace).
- Hay 2 maneras de crear agentes.
  - Primitives: Usando la graph api o la functional api
    Los agentes desde 0
  - PreBuilds: Son menos costosos porque ya vienen creados.

La api graph cuenta de 3 partes

- State
- Node
- Edges

Algunal del dia tu graph es un state engine.
El STATE son los datos que el agente actualiza en la operacion
Los estados (STATE) cuentan con 2 elementos principales,
-SCHEMA
- REDUCERS

Los schema es los elementos que vas a tener en STATE defines el modelo de datos que vas a tener.

Las Reducers es lo que aplica actualizaciones a los estados (STATE) y controlan estas actualizaciones.
Cada elemento del STATE tiene su Reducer.

Los STATE se representan como diccionarios puedes usar muchas lib para hacerlo.

Hay 3 maneras de definir tu STATE schema

- diccionario
- TypeDict
- Pydantic + TypeDict

Observacion: TypeDict es mas performante que Pydantic, pero Pydantic te permite hacer validacion, tiene default set, etc. etc.

REDUCERS

When you go form one node to another you may want want to update thes add state variables and to do that you need a reducer. If you dont define any, there is a default reducer that will take action.
This default reducer what he does is override the state variable value with the new value.

One custom reducer is a function that takes 2 params, the current value and the new value that the Node returns.

To use a custom reducer you need to use the Annotated class from Typing when yo give the type on the custom reduce function

Annotated [list, custom_reducer]

Langgraph provides a special reducer called add_message for handleing chat history with LLM interactions.

add_messages REDUCER

The main Purpose of this reducer is to help yo manage LLM conversations/messages
We need to maintain context an manag this conversations correctly

You can use this reduce to handel this type of messages:

- HumanMessage
- BaseMessage
- AIMessage
- ToolMessage
- SystemMessage (Just one per each workflow to give instructions to the LLM)

Bbase

- BaseMessage is the by class that holds the diferent type of Messages, so when use add_messages we need to set the type with BaseMessage

  · messages : Annotated [List [BaseMessage], add_messages]

- MessagesState allows you to dont have to write the goin green code in the TypeDic class al the time y can do:

```
class AgentState (MessagesState):
    (No need of messages: Annotated [----.
    Start adding the variable state yo need.
    step_count : int.
```

NODES

They are the action takers in Langgraph.
You can call them in seccuance or in parallel.
All the proces run in Nodes (llm calls, api calls...)
They also represents STEPS in a particular workflow.

Langgraph converts nodes in RunableLamda to execute.

Runble Lamda is a function wrapped in a standard interface so we can compose it, chain, retry, run in parallel, trace...

Core IDEA

```
cont result = myFunction (input)
```

Whit Runable Lamdas is:

```
const runable = RunnableLamda.from (myFunction)
const result = await runnable.invoke (input)
```

~~What~~ With ↑ we can trace, retry, parallelized runns, observe, stream, chain with other steps.

Typicly NODE take a state an retur a updates state.

* Only return the state variables that y want to update.

There are more things that you cantake as a param in a Node.

- Config
- Context_schema. (we usually work with context schema rather than config)
- runtime

EDGES.

This ones define the flow of the workflow, they are the routing logic.
You can move from one node to another directlly or conditonally.

Basicly they are the orchestrator.

- Normal Edges (From 1 point to another)
- Conditional Edges. (Move from one node to another dynamicly based in condition)

RUNTIM CONTEXT.

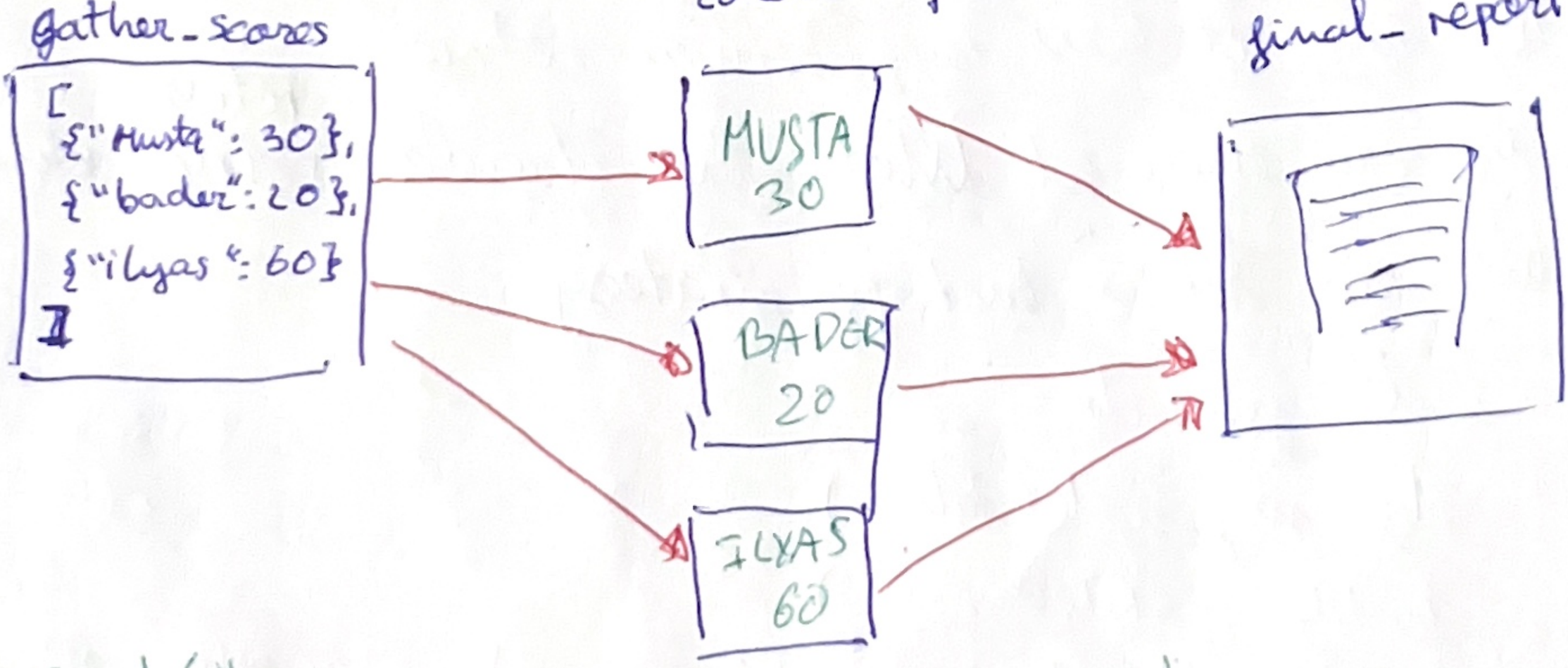
When you have some values that you dont want to specify in your states. Mostly because they dont change from one node to another like configurations, like DTATABASE_URL, LLM PROVIDER, PROMPTS.... you add them in the Runtim Context.

- You provide values at the runtime.
- they cannot be modify by your node
- Can be override at your invocation.
- When set the schema you can provide dfault values
- Use @dataclass as schema class as contextschema.

Send()

Ex: Is the MAP REDUCE that allow us to run parallel nodes with private data and gather the result in a final node:

Example:
- Imagine that you have a node that gather all user scors
- Then you have a node that calculate the grade based on that score
- And at the end send everything to a final-report node that gather all the grades.



send("calculate_grade", {"score": 60})

Allows you to run in parallel the calculate in this case...

Retry Policy.

This policy allow you to have resilient Nodes that can selfheal from issues an allow you to retray.

Imagin you what to send an email an the NODE that send the email fails with a 500 error code you what the NODE to retry.

- We can set the amount of time that we want it to retry
- We can set how long between retries
- We can even set what erros should retry on
- We can set a backoff strategy.

To use it you do:

```
def my-node(state...)
builder = StateGraph(...)
builder.add_node("my-node", my-node, retry-policy = RetryPolicy())
                                                      ↑
                                                    class
```

↳ If i use it like that it will use default retry policy

```
max_attempts int
initial_interval float
backoff_factor float (Is a multiplayer.)
max_interval float (limits the backoff)
jitter bool (avoid nodes with same retry policy try in same time)
retry_on = (TimeoutError, ConectionError, RateLimitError) tuple.
```