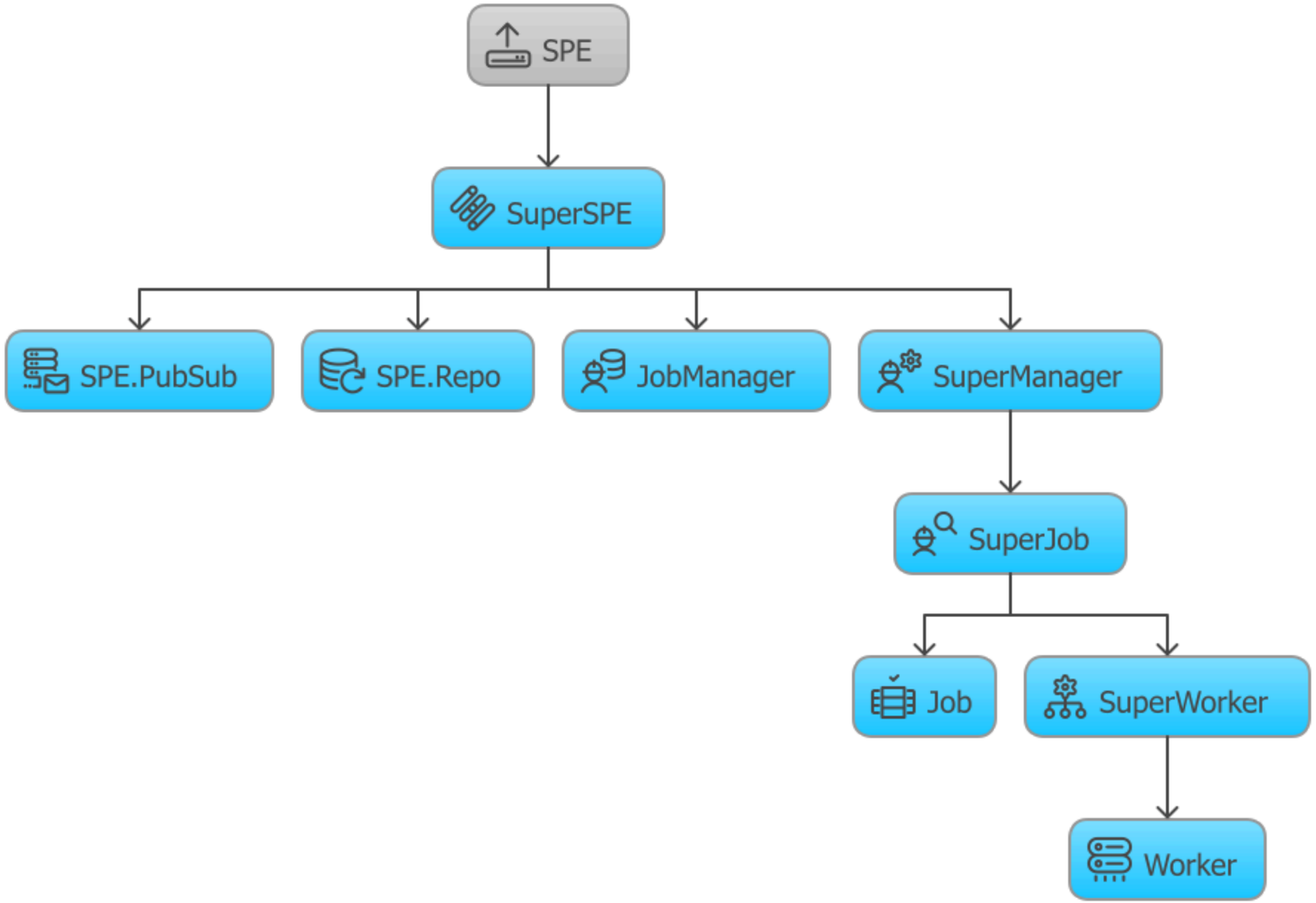
# SPE: Job Processing Engine


Lucas Rus Gallego
l.rus@alumnos.upm.es

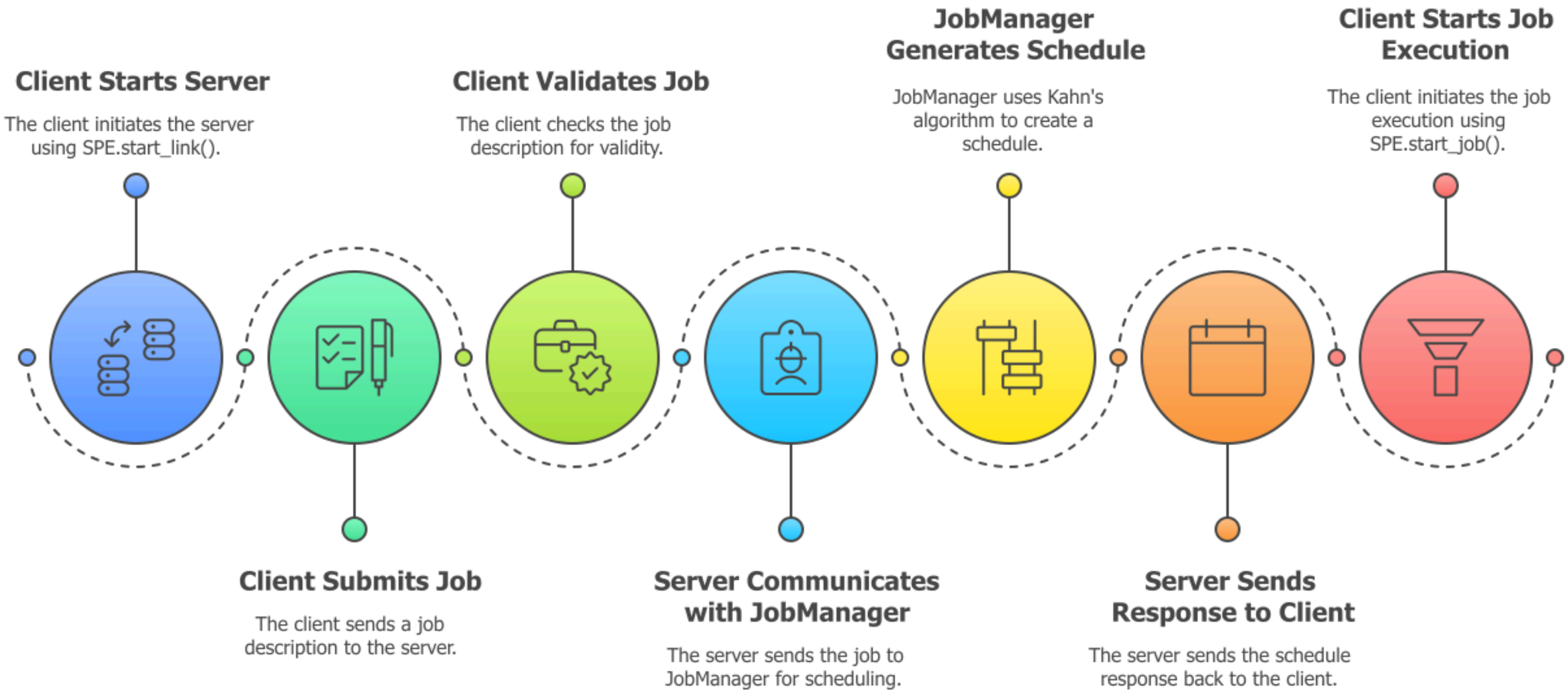
Alejandro Ortega Hernández
al.ortega@alumnos.upm.es

Eduardo Gil Alba
eduardo.gil.alba@alumnos.upm.es


# Architecture Design



# SPE Workflow



# Supervising

Why do we need to use Supervisor?

In case of failure, ensures one crashing process won´t take down the entire system. Even if it can´t be restarted, the failure is isolated, keeping the rest stable.

The top-level supervisor, **SuperSPE**, oversees the entire system.

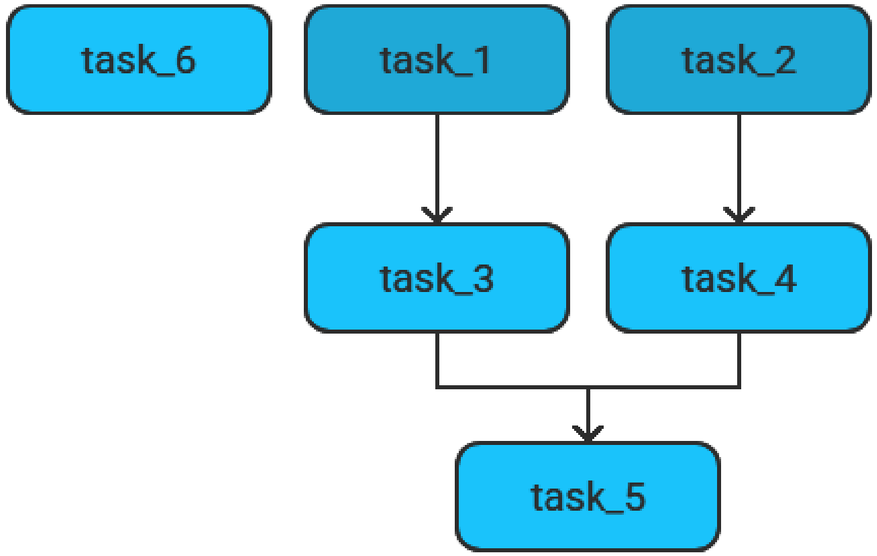
**SuperManager** supervisor dynamically spawns additional child supervisors whenever a start job request is initiated.

Each of these child supervisors (**SuperJob**) contains:

- A **Job** server to coordinate task progress and delegate work to Workers.
- A **SuperWorker** supervisor, which manages the lifecycle of the Worker processes that execute tasks.

# Scheduling

**Planner** perform the scheduling following **Kahn's algorithm**, grouping the resulting ordered tasks in lists whith possible concurrent task, taking **num_workers** into account




Kahn´s Output

```
kahns = [task_1, task_2, task_6, task_3, task_4, task_5]
```

---

```
if num_workers = 2
```

↓

Schedule

```
[[task_1, task_2], [task_6, task_3], [task_4], [task_5]]
```

---

```
if num_workers = :unbound
```

↓ `num_workers = length(kahns) = 6`

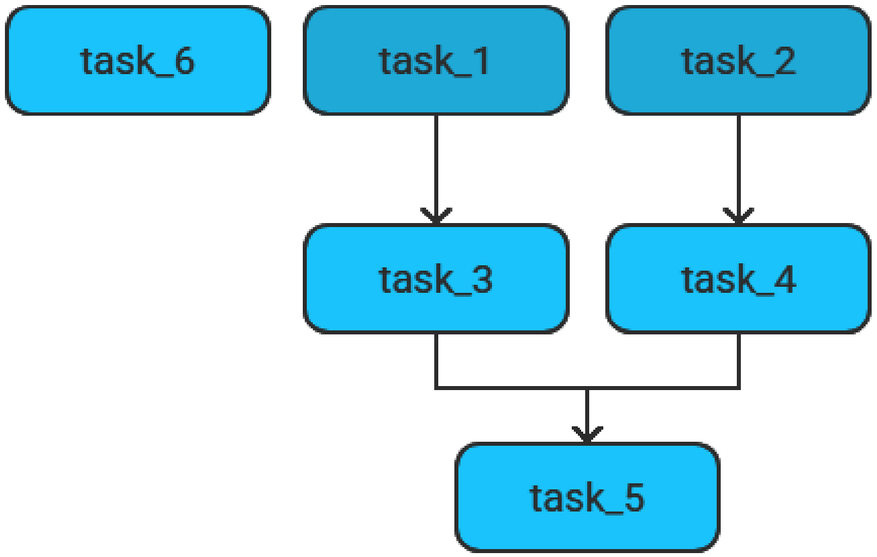
Schedule

```
[[task_1, task_2, task_6], [task_3, task_4], [task_5]]
```

In **JobManager** if **num_workers** exceeds the length of the longest possible concurrent task list, it's readjusted to that length to avoid costly process creation

# Optional: Task Priority

In job desciption could be provided a parameter **priority** as a list of tasks that should be runned first. When grouping tasks into sub-lists (respecting concurrency limits), these high-priority tasks are placed at the front.




Kahn´s Output

```
kahns = [task_1, task_2, task_6, task_3, task_4, task_5]
```

```
if num_workers = 2 && priority = [task_2, task_4]
```

↓

Schedule

```
[[task_2, task_1], [task_4, task_6], [task_3], [task_5]]
```

```
if num_workers = :unbound && priority = [task_2, task_4]
```

↓ num_workers = length(kahns) = 6

Schedule

```
[[task_2, task_1, task_6], [task_4, task_3], [task_5]]
```

If **num_workers** is **:unbound** the same agroupation will be made also taking **priority** into account placing the selected tasks as far forward in the lists as possible

# Monitoring

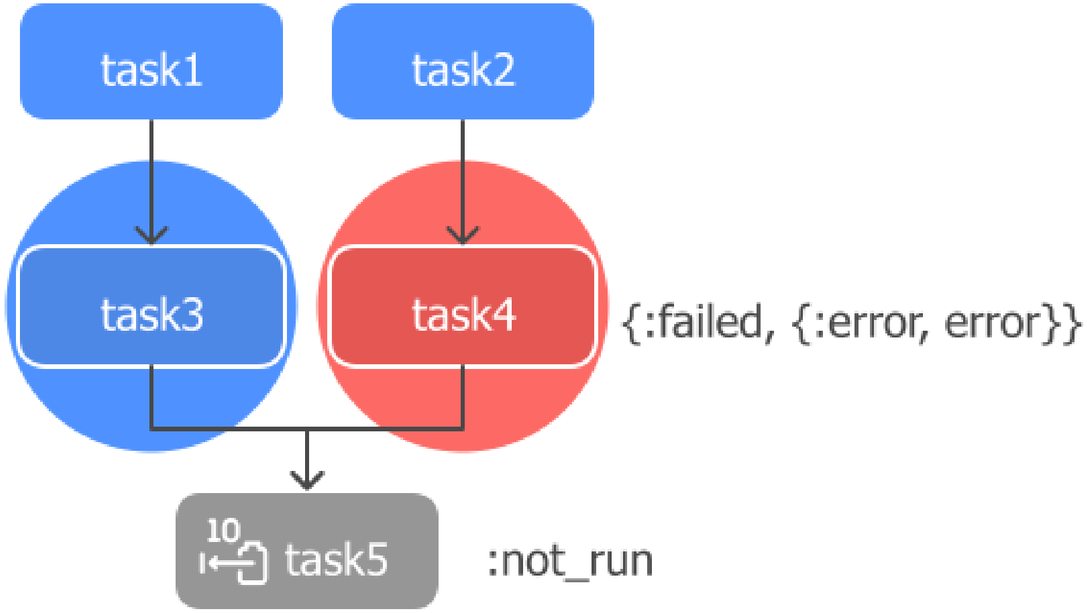
When a **Worker** completes a task, it notifies its parent **Job** via callback.

The **Job** then assigns the next available task until the schedule is empty.

If a **Worker** crashes:

- The **Job** receives

  **{:DOWN, _ref, :process, pid, reason}**
- Identifies the interrupted task
- Analyze enables and performs cleanup by:
  - Removing affected tasks from the schedule
  - Marking them as :not_run for potential retries



# Worker

**Responsibility**

Individual tasks execution

**Safe Task Execution**

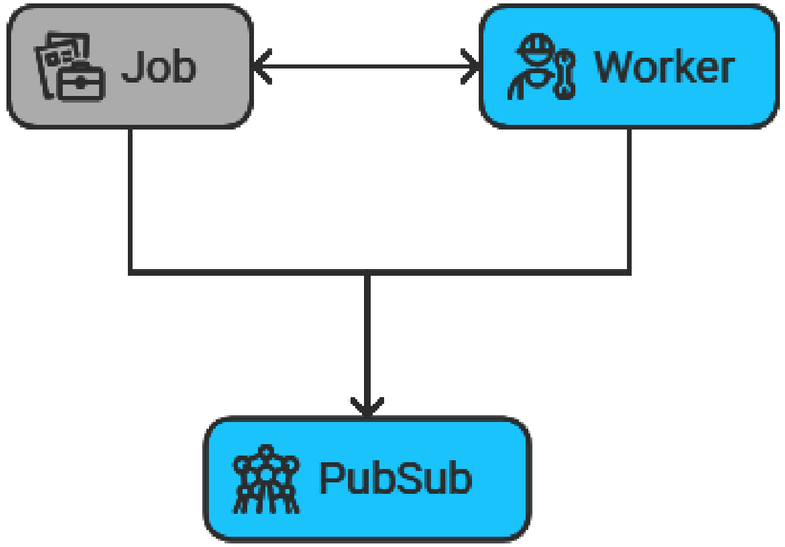
Usage of Task.async and Task.await

**Communication**

Notifies task status via PubSub messages

**Exception handling**

Usage of try/rescue
Returning safe results:
{:failed, {:crashed, reason}}
{:failed, :timeout}



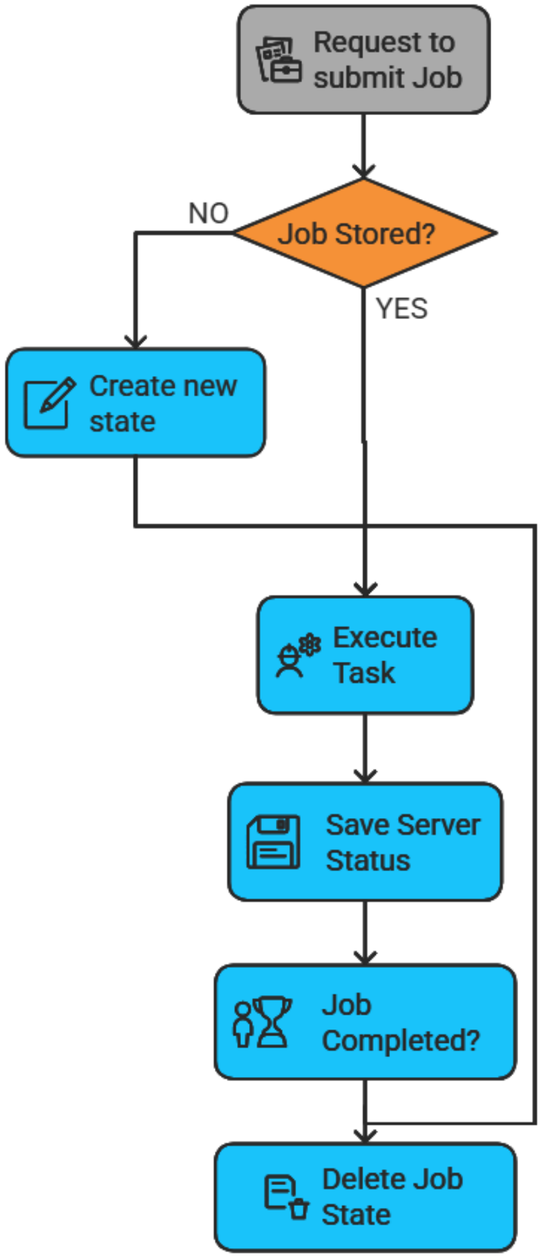
# Optional: Persistence

- Each time a submit request is received, it will examine if the job is stored and retrieve the state.
- Either if it is stored or not, it will redefine a new schedule for tasks execution.

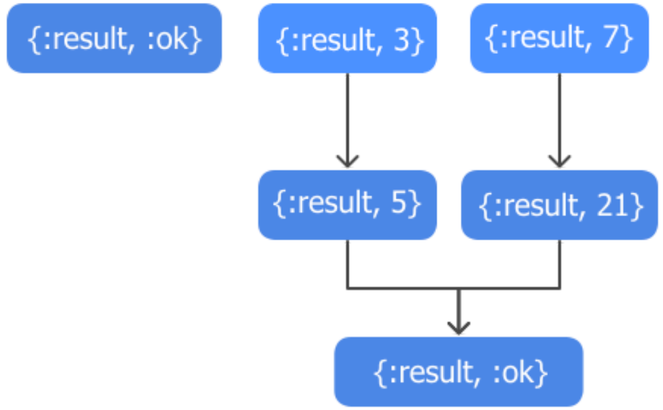
Implementing **Ecto.Repo** allow us to establish the connection with RDBMS.

**JobRepository** will be used to perform queries supported by a well defined job state schema.

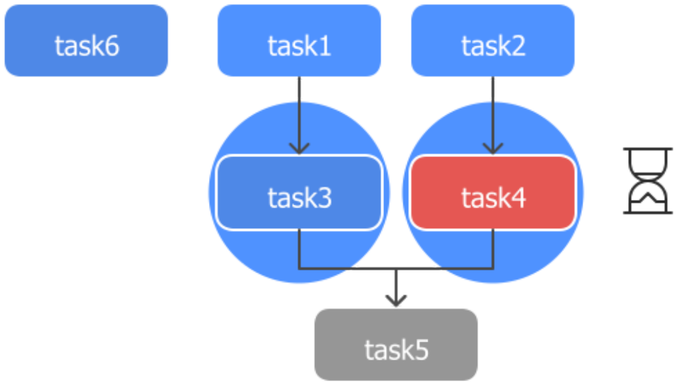
An auxiliar module, **Serializer**, is used to wrap the serialization logic.



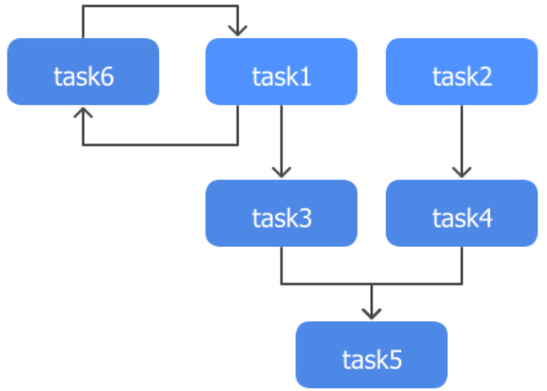
# Testing I



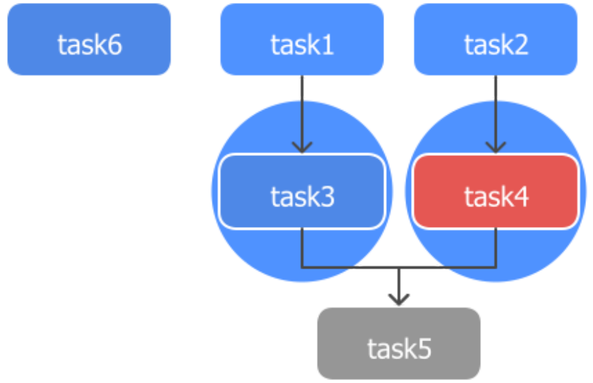

```
{:succeeded, {“task6” =>{:result, :ok},
      “task1” =>{:result, 3} ...}
```




```
{:failed, :timeout}
```

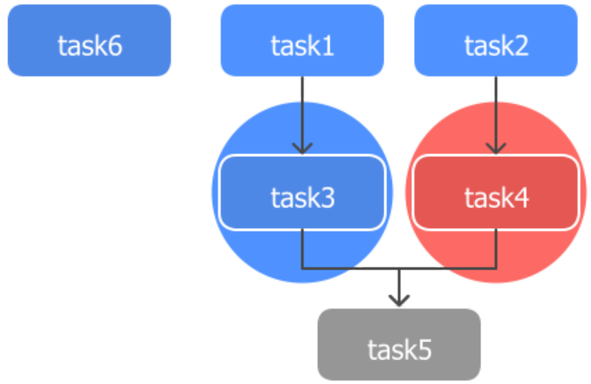



```
{:error, :graph_has_cycle}
```

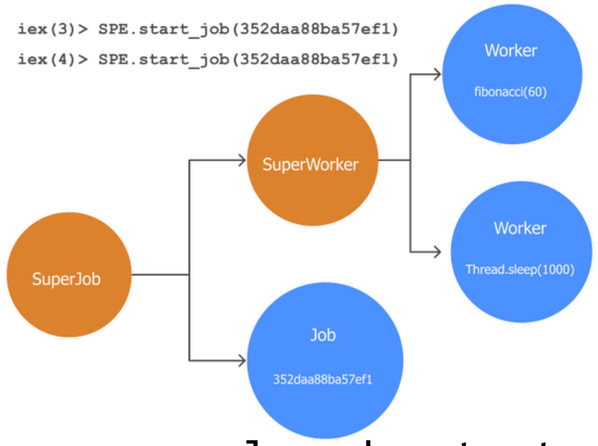



```
{:failed, {:crashed, "no_future"}}
```

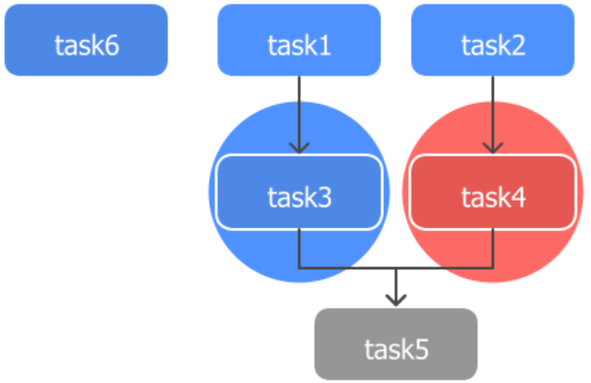
# Testing II




```
{:failed, {:crashed, {{:nocatch, "dont
    catch me"}, __STACKTRACE__}}
```




```
{:error, :already_started}
```




```
{:failed, {:crashed, :brutal_kill}}
```

```
task1 = %{
   "name" => "t1",
   "enables" => [""],
   "exec" => fn x, y -> 3 / 0 end,
   "timeout" => -100
}
```

```
{:error, :invalid_description}
```

# Developing Challenges

Scheduling jobs

Is it already decided? Or it will decide it at the moment? Both?
What if the number of workers is not defined? Do we need all of this workers?

Process Synchronization

Is it the Worker ready? Should I continue? Did the task either fail or completed?

Design the infrastucture to avoid a Worker failure take down the server

How do the Job noticed if a Worker still alive? How to “revive” them?

Serializing data structures (tuples)

# Project Outcomes: Key Learnings

- **Robustness & Fault Tolerance**: Elixir/OTP's "let it crash" philosophy, through GenServers and Supervisors, is crucial for automatic crash recovery and building stable, fault-tolerant systems.

- **Asynchronous Communication**: Phoenix.PubSub was essential for decoupling task and job result reporting, a vital pattern for scalable and distributed architectures.

- **Effective Debugging**: The Logger library was invaluable for understanding system behavior and efficiently debugging in a scalable environment.

# Thank you!

Any Questions?

Lucas Rus Gallego
l.rus@alumnos.upm.es

Alejandro Ortega Hernández
al.ortega@alumnos.upm.es

Eduardo Gil Alba
eduardo.gil.alba@alumnos.upm.es