Below is a comprehensive, advanced setup manager script that combines **external configuration**, **interactive live monitoring**, **advanced dependency management**, and even **parallel execution**—all in one. In addition, it automatically generates a PlantUML diagram of the process and includes your project management guide file (PM-tool.md) for reference. This design follows the latest industry trends for project orchestration and maintainability.

Below you’ll find two key parts:

1. **An external YAML configuration file (e.g.** tasks.yaml**)** This file lets non-developers easily adjust the setup order, add new tasks or set dependencies.
2. **The advanced Python orchestrator (**setup\_manager\_advanced.py**)** This script:
   * Reads the tasks from the YAML file.
   * Uses Rich to display live progress and status updates.
   * Uses a scheduler that examines dependencies between tasks. Tasks whose dependencies are already completed are launched concurrently. Tasks whose dependencies fail are automatically skipped.
   * Generates a PlantUML file (and if available, a PNG diagram) for a visual, non-technical explanation of the process.
   * Logs all output to an advanced log file.

**1. Sample Configuration File (tasks.yaml)**

Create a file named tasks.yaml (in the same directory as the script) with content similar to:

yaml

tasks:

- name: "Setup Environment"

path: "C:\\Users\\ACER\\Documents\\BEunixUb - Setup Standard\\setup\_environment.py"

type: "py"

- name: "Setup Structure"

path: "C:\\Users\\ACER\\Documents\\BEunixUb - Setup Standard\\setup\_structure.py"

type: "py"

dependencies:

- "Setup Environment"

- name: "Apache Airflow Setup"

path: "C:\\Users\\ACER\\Documents\\BEunixUb - Setup Standard\\Apache Airflow.py"

type: "py"

dependencies:

- "Setup Environment"

- name: "PM Tool Guidance"

path: "C:\\Users\\ACER\\Documents\\BEunixUb - Setup Standard\\PM-tool.md"

type: "doc"

# … add as many tasks as you wish, including your add-ons.

You may of course add additional tasks (such as EXE or BAT files, API integrations, or additional Python scripts) with an optional dependencies key to indicate prerequisite tasks.

**2. Advanced Setup Manager Script (setup\_manager\_advanced.py)**

Save the following script as setup\_manager\_advanced.py and adjust as needed. (Note: on Ubuntu you may have to skip Windows-specific files unless you’re using Wine.)

python

#!/usr/bin/env python3

import os

import subprocess

import logging

import platform

import time

import sys

import yaml

import shutil

from concurrent.futures import ThreadPoolExecutor, as\_completed

from rich.console import Console

from rich.table import Table

from rich.progress import Progress, SpinnerColumn, BarColumn, TextColumn

# ------------------------------------------------------------------------------

# Logging Configuration

# ------------------------------------------------------------------------------

logging.basicConfig(

filename="setup\_manager\_advanced.log",

level=logging.INFO,

format="%(asctime)s - %(levelname)s - %(message)s",

)

console = Console()

CONFIG\_FILE = "tasks.yaml" # External configuration file

# ------------------------------------------------------------------------------

# Function: load\_tasks

# Reads tasks from a YAML config file.

# ------------------------------------------------------------------------------

def load\_tasks(config\_file):

if not os.path.exists(config\_file):

console.print(f"[Error] Config file '{config\_file}' not found.", style="bold red")

sys.exit(1)

with open(config\_file, "r") as f:

config = yaml.safe\_load(f)

return config.get("tasks", [])

# ------------------------------------------------------------------------------

# Function: run\_task

# Executes a task based on its type.

#

# Supported types:

# - "py": Python script executed using python3.

# - "sh": Shell script executed with bash.

# - "bat": Batch file (only on Windows).

# - "exe": Executable (or via Wine on Linux).

# - "doc": Documentation file, skipped (but listed in the diagram).

# ------------------------------------------------------------------------------

def run\_task(task):

task\_name = task.get("name")

file\_path = task.get("path")

file\_type = task.get("type")

if not os.path.exists(file\_path):

msg = f"File not found for task '{task\_name}': {file\_path}"

logging.error(msg)

console.print(f"[Error] {msg}", style="red")

return False

sys\_platform = platform.system()

command = None

if file\_type == "py":

command = f"python3 \"{file\_path}\""

elif file\_type == "sh":

command = f"bash \"{file\_path}\""

elif file\_type == "bat":

if sys\_platform == "Windows":

command = f"\"{file\_path}\""

else:

logging.warning(f"Skipping BAT task '{task\_name}' on {sys\_platform}.")

console.print(f"[Warning] Skipping {task\_name} (BAT file not supported on {sys\_platform}).", style="yellow")

return True

elif file\_type == "exe":

if sys\_platform == "Windows":

command = f"\"{file\_path}\""

else:

if shutil.which("wine"):

command = f"wine \"{file\_path}\""

else:

logging.warning(f"Skipping EXE task '{task\_name}' on {sys\_platform} without Wine.")

console.print(f"[Warning] Skipping {task\_name} (EXE file not supported on {sys\_platform} without Wine).", style="yellow")

return True

elif file\_type == "doc":

# Documentation file. Not executed, but included in the diagram.

logging.info(f"Skipping document task '{task\_name}'.")

console.print(f"[Info] Skipping {task\_name} (document).", style="blue")

return True

else:

logging.error(f"Unknown file type for task '{task\_name}'.")

console.print(f"[Error] Unknown file type for {task\_name}.", style="red")

return False

try:

logging.info(f"Executing task '{task\_name}' with command: {command}")

result = subprocess.run(command, shell=True, capture\_output=True, text=True)

if result.returncode != 0:

logging.error(f"Task '{task\_name}' failed: {result.stderr}")

console.print(f"[Error] Task '{task\_name}' failed.", style="red")

return False

else:

logging.info(f"Task '{task\_name}' completed successfully.")

console.print(f"[Completed] {task\_name}", style="green")

return True

except Exception as e:

logging.exception(f"Exception in task '{task\_name}': {str(e)}")

console.print(f"[Exception] Task '{task\_name}' encountered an exception.", style="red")

return False

# ------------------------------------------------------------------------------

# Function: generate\_plantuml\_diagram

# Writes a PlantUML file mapping the entire setup process.

# ------------------------------------------------------------------------------

def generate\_plantuml\_diagram(tasks, output\_file="setup\_process\_advanced.puml"):

with open(output\_file, "w") as f:

f.write("@startuml\n")

f.write("title BEunixCsuite Advanced Setup Process\n\n")

for idx, task in enumerate(tasks):

f.write(f"rectangle \"{task.get('name')}\" as T{idx+1}\n")

f.write("\n' Define sequential flow (for illustration only)\n")

for idx in range(len(tasks) - 1):

f.write(f"T{idx+1} --> T{idx+2}\n")

f.write("@enduml\n")

console.print(f"[Diagram] Generated PlantUML file: {output\_file}", style="cyan")

if shutil.which("plantuml"):

try:

subprocess.run(["plantuml", output\_file], check=True)

console.print("[Diagram] Generated PNG diagram using PlantUML.", style="cyan")

except subprocess.CalledProcessError as e:

console.print(f"[Diagram Error] {e}", style="red")

else:

console.print("[Diagram] PlantUML not found. Install it to auto-generate diagrams.", style="yellow")

# ------------------------------------------------------------------------------

# Function: advanced\_scheduler

#

# This scheduler:

# • Builds a dependency graph.

# • Runs tasks concurrently when all their dependencies are met.

# • Automatically skips tasks if any dependency fails.

# • Provides live interactive progress updates using Rich.

# ------------------------------------------------------------------------------

def advanced\_scheduler(tasks):

# Map tasks by name for quick lookup.

task\_dict = {task["name"]: task for task in tasks}

# Outcomes will hold the status: "success", "failed", or "skipped (failed dependency)"

outcomes = {}

pending = set(task\_dict.keys())

progress = Progress(

SpinnerColumn(),

TextColumn("[progress.description]{task.description}"),

BarColumn(),

TextColumn("{task.percentage:>3.0f}%"),

)

progress\_task = progress.add\_task("[bold green]Executing tasks...", total=len(pending))

console.print("\n[Starting] Advanced Setup Process...\n", style="bold underline")

with progress:

while pending:

ready = []

# Iterate over a copy of pending tasks.

for task\_name in list(pending):

task = task\_dict[task\_name]

dependencies = task.get("dependencies", [])

# If dependencies are not yet completed, skip for now.

if any(dep not in outcomes for dep in dependencies):

continue

# If any dependency did not succeed, mark this task as skipped.

if any(outcomes.get(dep) != "success" for dep in dependencies):

outcomes[task\_name] = "skipped (failed dependency)"

pending.remove(task\_name)

logging.warning(f"Skipping '{task\_name}' due to failed dependency in {dependencies}.")

console.print(f"[Skip] {task\_name} (dependency failure)", style="yellow")

progress.advance(progress\_task)

else:

ready.append(task\_name)

# If no tasks are ready and some are still pending, wait briefly.

if not ready:

time.sleep(0.5)

continue

# Launch ready tasks in parallel.

with ThreadPoolExecutor(max\_workers=len(ready)) as executor:

future\_to\_task = {

executor.submit(run\_task, task\_dict[task\_name]): task\_name for task\_name in ready

}

for future in as\_completed(future\_to\_task):

task\_name = future\_to\_task[future]

try:

result = future.result()

outcomes[task\_name] = "success" if result else "failed"

except Exception as exc:

logging.exception(f"Task '{task\_name}' exception: {exc}")

outcomes[task\_name] = "failed"

pending.discard(task\_name)

progress.advance(progress\_task)

# Display a summary table after processing all tasks.

table = Table(title="Setup Process Summary")

table.add\_column("Task", style="cyan", no\_wrap=True)

table.add\_column("Outcome", style="magenta")

for task\_name in task\_dict:

outcome = outcomes.get(task\_name, "skipped (not executed)")

table.add\_row(task\_name, outcome)

console.print(table)

return outcomes

# ------------------------------------------------------------------------------

# Main Execution

# ------------------------------------------------------------------------------

def main():

tasks = load\_tasks(CONFIG\_FILE)

# Generate a PlantUML diagram for a high-level view.

generate\_plantuml\_diagram(tasks)

# Run tasks using the advanced scheduler (dependency management & parallel execution)

outcomes = advanced\_scheduler(tasks)

console.print("\n[Complete] Setup process finished. Check 'setup\_manager\_advanced.log' for details.", style="bold green")

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Next Steps & Further Ideas**

* **Configuration File:** By moving the task definitions to an external YAML file, non-developers can easily reorder tasks and set dependencies without changing code.
* **Interactive Monitoring:** We’re using the Rich library to display live progress bars and status updates in a stylish, easy-to-read format for technical and non-technical audiences alike.
* **Advanced Dependency Management:** The scheduler checks each task’s dependencies. If a dependency fails, any task needing it is automatically skipped. This prevents unnecessary errors and clarifies which steps require remediation.
* **Parallel Execution:** Tasks that are independent (or whose dependencies have been met) run in parallel to accelerate the setup process.
* **Project Management Guidance:** The PM-tool.md file is integrated as a documentation task in the configuration. It contains comprehensive guidance and the latest standards in project management. Stakeholders can review this file (outside of execution) to understand the project’s roadmap and process improvements.

**Requirements & Installation**

Make sure to install the required Python packages before running the script:

bash

pip install pyyaml rich

And if you plan to generate diagrams with PlantUML (optionally):

* Install PlantUML and ensure it’s available on your system's PATH.
* (On Ubuntu, you might need to install Java if not already installed.)

This advanced setup manager serves as a robust backbone for your **BEunixCsuite** project—providing clarity for both technical execution and stakeholder transparency. Enjoy experimenting with these modern orchestration practices, and feel free to expand on dependency rules or the interactive dashboard as your project grows!