Asynchronous Distributed Plugin Execution Engine with AST Sandboxing and Metaclass Registry

Keywords: advanced python, asyncio, distributed computing, plugin architecture, AST sandboxing, multiprocessing, metaclasses, generic typing, python decorators, dynamic execution, tech giant level, interview ready, HR portfolio, concurrency, parallelism, code injection protection, machine-safe execution.

```
"""

Distributed Plugin Execution Engine (DPEE)

Tech-Giant-Ready Advanced Python Example

Featuring AST sandboxing, asyncio, multiprocessing, metaclasses, decorators, dynamic execution, and more

"""


import ast

import asyncio

import inspect

import multiprocessing

import time

from abc import ABC, abstractmethod

from asyncio import Queue

from types import ModuleType

from typing import Dict, Any, Callable, List, Type, Generic, TypeVar, Optional
```

```
# ======================
# AST-Based Sandbox
# ======================

class SafeEvaluator(ast.NodeVisitor):
    """
    Evaluates AST safely by only allowing specific node types.
    """

    SAFE_NODES = (
        ast.Module, ast.Expr, ast.Load,
        ast.Call, ast.Name, ast.BinOp,
        ast.Num, ast.Str, ast.UnaryOp,
        ast.Add, ast.Sub, ast.Mult, ast.Div,
        ast.Pow, ast.Assign, ast.Compare,
        ast.IfExp, ast.BoolOp, ast.And, ast.Or,
        ast.Eq, ast.NotEq, ast.Lt, ast.Gt,
        ast.Lambda
    )

    def generic_visit(self, node):
        if not isinstance(node, self.SAFE_NODES):
            raise ValueError(f"Disallowed node: {type(node).__name__}")
        super().generic_visit(node)
```

```
def safe_exec(code: str, context: Dict[str, Any]):
    """
    Execute code string securely using AST sandbox.
    """
    tree = ast.parse(code)
    SafeEvaluator().visit(tree)
    compiled = compile(tree, "<sandbox>", "exec")
    exec(compiled, context)

# ========================
# Plugin Metaclass
# ========================

class PluginMeta(type):
    """
    Registers all plugin classes automatically.
    """
    registry: Dict[str, Type['BasePlugin']] = {}

    def __new__(mcs, name, bases, namespace):
        cls = super().__new__(mcs, name, bases, namespace)
        if not name.startswith('Base'):
            PluginMeta.registry[name] = cls
        return cls
```

```
# ========================
# Abstract Base Plugin
# ========================

T = TypeVar("T")

class BasePlugin(ABC, Generic[T], metaclass=PluginMeta):
    @abstractmethod
    def run(self, input_data: T) -> Any:
        """
        Run the plugin with input and return processed output.
        """
        pass

# ========================
# Sample Plugin: ML Preprocessing
# ========================

class NormalizePlugin(BasePlugin[List[float]]):
    """
    Normalizes a list of floats between 0 and 1.
    """
    def run(self, input_data: List[float]) -> List[float]:
        min_val = min(input_data)
        max_val = max(input_data)
        return [(x - min_val) / (max_val - min_val + 1e-10) for x in input_data]


class ReverseTextPlugin(BasePlugin[str]):
    """
    Reverses the given string.
    """
    def run(self, input_data: str) -> str:
        return input_data[::-1]


# ========================
# Asynchronous Plugin Executor
# ========================


class AsyncExecutor:
    """
    Dispatches plugin tasks asynchronously.
    """


    def __init__(self):
        self.task_queue: Queue = Queue()
        self.results: List[Any] = []


    async def enqueue(self, plugin: BasePlugin, data: Any):
        await self.task_queue.put((plugin, data))
    async def worker(self):
        while True:
            plugin, data = await self.task_queue.get()
            if plugin is None:
                break
            result = await asyncio.to_thread(plugin.run, data)
            self.results.append((plugin.__class__.__name__, result))
            self.task_queue.task_done()

    async def run_workers(self, n: int = 2):
        workers = [asyncio.create_task(self.worker()) for _ in range(n)]
        await self.task_queue.join()
        for _ in workers:
            await self.task_queue.put((None, None))  # Sentinel
        await asyncio.gather(*workers)

# ========================
#Multiprocessing Safe Isolated Execution
# ========================

def safe_plugin_process(plugin_class: Type[BasePlugin], data: Any, result_queue: multiprocessing.Queue):
    """
    Execute plugin in isolated process for safety.
    """
    try:
        plugin = plugin_class()
        result = plugin.run(data)
        result_queue.put((plugin_class.__name__, result))
    except Exception as e:
        result_queue.put((plugin_class.__name__, f"Error: {e}"))

def run_in_process(plugin_class: Type[BasePlugin], data: Any):
    q = multiprocessing.Queue()
    p = multiprocessing.Process(target=safe_plugin_process, args=(plugin_class, data, q))
    p.start()
    p.join(timeout=5)
    if p.is_alive():
        p.terminate()
    return q.get()

# ========================
# Demo and Test Harness
# ========================

async def main():
    print("Initializing Plugin Execution Engine...")

    executor = AsyncExecutor()
    await executor.enqueue(NormalizePlugin(), [5.0, 15.0, 30.0])
    await executor.enqueue(ReverseTextPlugin(), "Tech Giant Ready Code")

    await executor.run_workers()

    print("\n Asynchronous Results:")
    for plugin_name, result in executor.results:
        print(f"{plugin_name}: {result}")

    print("\n Running in Isolated Process:")
    output = run_in_process(NormalizePlugin, [10.0, 20.0, 40.0])
    print(f" NormalizePlugin (Isolated): {output}")

    print("\n Executing sandboxed user code:")
    user_code = "result = sum([1, 2, 3, 4])"
    context = {}
    safe_exec(user_code, context)
    print(f" Sandboxed Result: {context['result']}")

if __name__ == "__main__":
    asyncio.run(main())
```