**Optimized Task Execution Code with Debugging and Enhancements**

The existing code lacked exception handling, logging, and retry mechanisms, leading to potential system failures. Below is the improved implementation incorporating best practices.

**Refactored Code with Improvements**

using System;

using System.Collections.Generic;

using System.Threading;

public class TaskExecutor

{

private Queue<string> taskQueue = new Queue<string>();

// Method to add a task to the queue

public void AddTask(string task)

{

if (string.IsNullOrWhiteSpace(task))

{

Console.WriteLine("Invalid task: Task cannot be null or empty.");

return;

}

taskQueue.Enqueue(task);

Console.WriteLine($"Task '{task}' added to queue.");

}

// Method to process tasks with error handling and retry mechanism

public void ProcessTasks()

{

while (taskQueue.Count > 0)

{

string task = taskQueue.Dequeue();

int retryCount = 3;

for (int attempt = 1; attempt <= retryCount; attempt++)

{

try

{

ExecuteTask(task);

Console.WriteLine($"Task '{task}' completed successfully.");

break; // Exit retry loop on success

}

catch (Exception ex)

{

Console.WriteLine($"Error processing task '{task}' (Attempt {attempt}/{retryCount}): {ex.Message}");

if (attempt == retryCount)

{

Console.WriteLine($"Task '{task}' failed after {retryCount} attempts.");

}

else

{

Thread.Sleep(1000); // Delay before retrying

}

}

}

}

}

// Simulated task execution method

private void ExecuteTask(string task)

{

if (new Random().Next(0, 4) == 0) // Simulated failure scenario (25% failure rate)

{

throw new Exception("Random execution failure.");

}

// Simulate task processing

Console.WriteLine($"Processing task: {task}");

}

// Main method to demonstrate task processing

public static void Main()

{

TaskExecutor executor = new TaskExecutor();

executor.AddTask("Task 1");

executor.AddTask("Task 2");

executor.AddTask(""); // Invalid task test

executor.AddTask("Task 3");

executor.ProcessTasks();

}

}

**Key Improvements & Explanations**

1. **Null & Empty Input Handling**
   * Previously, AddTask did not check for null or empty tasks.
   * Now, it validates input and prevents invalid tasks from being added.
2. **Exception Handling**
   * The previous implementation lacked try-catch blocks.
   * Introduced structured exception handling in ProcessTasks() to prevent crashes.
3. **Retry Mechanism for Failures**
   * If a task fails, the system retries it up to **3 times** before giving up.
   * A **1-second delay** (Thread.Sleep(1000)) prevents immediate reattempts.
4. **Logging for Better Debugging**
   * Instead of just printing task execution, we log **success and failure messages**.
   * If a task fails after multiple retries, an appropriate message is displayed.

**Reflection on LLM Assistance**

**1. How did the LLM assist in debugging and optimizing the code?**

* The LLM identified **key areas of failure**: lack of exception handling, missing input validation, and inefficiencies in handling failed tasks.
* Suggested adding **error logging and retry logic** to prevent crashes.
* Improved **readability** by breaking down logic into separate methods.

**2. Were any LLM-generated suggestions inaccurate or unnecessary?**

* Some suggestions involved using **complex concurrency** (e.g., multi-threading), which wasn't necessary for a simple task queue.
* Overly verbose logging was suggested, which I simplified to essential error tracking.

**3. What were the most impactful improvements?**

* Implementing **retry logic** for tasks, preventing unnecessary failures.
* Adding **input validation** to ensure no invalid tasks are added.
* Proper **exception handling** to prevent system crashes.

This refactored version ensures that **SwiftCollab’s task execution system remains stable, efficient, and fault-tolerant.** 🚀