Architectural Flaws

1. Handling Multiple Clients (Multithreading)

- **Old Code:** The old server could only handle one client at a time, because it processed each client in the main loop without spawning separate threads. Once a client was connected, it blocked the server from accepting other connections until the current client disconnected.
- **New Code:** The new server spawns a new thread for each incoming client connection using thread::spawn. This allows the server to handle multiple clients concurrently. Each client is processed in a separate thread, which means the server can handle multiple client requests at the same time without blocking on any single client. The "is_running" flag is shared across threads safely using Arc<Mutex<AtomicBool>>, ensuring that threads can check if the server is still running before continuing operations.

Code snippet

```
// Spawn a new thread to handle the client independently
thread::spawn(move || {
    let mut client = Client::new(stream, is_running_clone);  // Create a new Clie
    client.handle(); // Call the `handle` method to process the client's requests
});
```

2. Correct Handling of WouldBlock Errors

- **Old Code:** In the old version, the code checked if the error kind was WouldBlock when accepting connections but did not do so when reading data from the client. This could result in high CPU usage because the program didn't efficiently handle cases where there was no data to read.
- **New Code:** The new version correctly checks for WouldBlock errors while reading from the client's stream. When no data is available, the server sleeps for 100ms to avoid busy-waiting and reduce CPU usage. This improves efficiency by ensuring that the server does not constantly poll for new data when there are no incoming messages, leading to a more efficient and scalable solution.

Code snippe

```
// Handle cases where no data is available yet
Err(ref e) if e.kind() == io::ErrorKind::WouldBlock => {
    // No data available, just return and retry later
    thread::sleep(Duration::from_millis(100)); // Sleep briefly to avoid busy waiting
}
```

3. Proper Synchronization for the "is_running" Flag

- **Old Code:** In the old version, the "is_running" flag was managed using an Arc<AtomicBool>, which allowed shared ownership of the flag but lacked proper synchronization mechanisms for concurrent access. This could lead to potential race conditions if multiple threads tried to modify or read the "is_running" flag simultaneously.
- **New Code:** The new version wraps the AtomicBool inside a Mutex, creating an Arc<Mutex<AtomicBool>>. This ensures that access to the "is_running" flag is properly synchronized across threads. Each thread locks the Mutex before reading or modifying the flag, ensuring that there are no race conditions and that the flag is safely shared across multiple threads.

Code snippet

```
let is_running = Arc::new(Mutex::new(AtomicBool::new(false)));
```

```
    {
        let is_running = self.is_running.lock().unwrap(); // Lock the Mutex to access is_running
        is_running.store(true, Ordering::SeqCst); // Mark the server as running
    }
```

4. Handling Different Types of Client Messages

- **Old Code:** The old version could only handle a basic EchoMessage type. If the server received any other message type, it failed to process it and didn't provide much feedback.
- **New Code:** The new version introduces a more flexible message-handling system by decoding different types of client messages. It uses pattern matching to distinguish between different message types such as AddRequest and EchoMessage. When an unknown message type is received, the server logs a warning instead of silently ignoring the message. This improves extensibility, allowing the server to easily handle new message types in the future without requiring major changes to the code.

Code snippet

```
match ClientMessage::decode(&buffer[..bytes_read]) {
    Ok(ClientMessage {
        message: Some(client_message::Message::AddRequest(add_request)),
    }) => {
        info!("Received AddRequest: a={}, b={}",add_request.a, add_request.b); // Log the request
        let result = add_request.a + add_request.b; // Perform the addition operation
        let response = ServerMessage {
            message: Some(server_message::Message::AddResponse(AddResponse {
                result,
            })),
        };
        let payload = response.encode_to_vec();
        if let Err(e) = self.stream.write_all(&payload) { // Handle any write errors
            error!("Error sending response: {}", e);
            break;
        }
        if let Err(e) = self.stream.flush() { // Ensure the data is flushed to the stream
            error!("Error flushing stream: {}", e);
            break;
        }
    }
```

```
Ok(ClientMessage {
    message: Some(client_message::Message::EchoMessage(echo_message)),
}) => {
    info!("Received EchoMessage: {}", echo_message.content); // Log the received message
    let response = ServerMessage {
        message: Some(server_message::Message::EchoMessage(EchoMessage {
            content: echo_message.content.clone(), // Echo back the same content
        })),
    };
    let payload = response.encode_to_vec();
    if let Err(e) = self.stream.write_all(&payload) { // Handle any write errors
        error!("Error sending response: {}", e);
        break;
    }
    if let Err(e) = self.stream.flush() { // Ensure the data is flushed to the stream
        error!("Error flushing stream: {}", e);
        break;
    }
}
Ok(_) => {
    warn!("Received unknown message type.");
}
Err(e) => {
    error!("Failed to decode message: {}", e);
```