Table of Contents

[1.0 Abstract 3](#_Toc168909284)

[2.0 Introduction and background 4](#_Toc168909285)

[2.1 Problem Statement 4](#_Toc168909286)

[3.0 Literature Review 6](#_Toc168909287)

[Introduction to Real-Time Systems and Rust 6](#_Toc168909288)

[Unique Features of Rust 6](#_Toc168909289)

[Concerns Addressed by Rust 8](#_Toc168909290)

[Linking Concepts in Rust 9](#_Toc168909291)

[Conclusion 10](#_Toc168909292)

[4.0 Design and Methodology 11](#_Toc168909293)

[Coding Implementation and Explanation 13](#_Toc168909294)

[5.0 Result and Discussion 29](#_Toc168909295)

[Disclaimer 29](#_Toc168909296)

[5.1 Benchmark 1 – Order Generation 29](#_Toc168909297)

[5.2 Benchmark 2 – Send Message 31](#_Toc168909298)

[5.3 Benchmark 3 – Receive Message 32](#_Toc168909299)

[5.4 Benchmark 4 – Inventory Check 34](#_Toc168909300)

[5.5 Benchmark 5 – Scheduled Inventory Report 35](#_Toc168909301)

[6.0 Conclusion 37](#_Toc168909302)

[7.0 References 38](#_Toc168909303)

# **Abstract**

This project presents a simulation of a warehouse management system (WMS) implemented in Rust, focusing on real-time communication and data processing. The system incorporates various components, including order processing, goods transportation, inventory management, and reporting. Through multithreading and communication channels, the system emulates real-world scenarios such as order generation, inventory updates, and scheduled reporting. Additionally, RabbitMQ messaging facilitates seamless communication between different subsystems. The project employs concurrency control mechanisms such as mutexes and leverages external libraries like amiquip for message queue handling. Benchmarking using bma\_benchmark and criterion provides insights into the performance of critical functions, aiding in optimization and scalability.

# **2.0 Introduction and background**

## 2.1 Problem Statement

Warehouses are essential for satisfying consumer needs and maximising operational effectiveness. Online shopping and worldwide supply chains have made warehouse operations more complex, which has prompted the deployment of automated technology to expedite procedures. Nevertheless, there are particular difficulties in developing and putting into practice these kinds of systems, such as the requirement for correct order processing, effective goods handling, and real-time inventory management. To address these challenges, this project delves into the realm of WMS, aiming to develop a simulation that encapsulates the complexities of warehouse operations. Leveraging the robustness of the Rust programming language and real-time computing concepts, the goal is to devise a solution that not only emulates real-world scenarios but also offers insights into optimizing warehouse processes.

The aim of this project is to create an intricate model that replicates the complexities of contemporary warehouse operations, simulating a Warehouse Management System (WMS). The simulation will cover a variety of subsystems with a variety of duties, including order fulfilment, inventory control, and logistics management.

**Challenges to address:**

1. Inventory Optimization: Effective management of inventory is essential for maximizing warehouse space utilization and ensuring product availability. The simulation must accurately model inventory tracking, replenishment strategies, and optimization techniques to minimize stock shortages and excess inventory.
2. Logistics Efficiency: Seamless movement of goods within the warehouse is crucial for optimizing workflow and minimizing operational delays. The simulation should simulate the transportation of goods between storage areas, picking stations, and shipping zones.
3. Task Coordination and Resource Management: Efficient coordination of warehouse tasks requires robust scheduling and resource allocation mechanisms. The simulation should incorporate advanced scheduling algorithms and concurrency strategies to optimize task execution and resource utilization while ensuring smooth operation and minimal downtime.

**Components/Subsystems of AWMS**

Inventory Management System: Real-time inventory tracking that monitors inventory levels and updates in real-time. Instantaneous inventory updates and uses dynamic inventory optimization.

Order Processing System: Processes incoming orders in real time. Immediate inventory validation to confirm product availability and prevent backorders or delays. Dynamic work order generation to optimize task allocation and resource utilization. Synchronized fulfilment integration that facilitates immediate order fulfilment and reducing order-to-delivery times.

Goods Transportation System: Real-time goods movement management, coordinates and controls the movement of goods within the warehouse.

# 3.0 Literature Review

## Introduction to Real-Time Systems and Rust

Real-Time Systems are essential to a variety of sectors including industrial automation, telecommunications, aerospace, and automotive. These systems have to comply with strict timeliness, reliability, and efficiency standards since they constantly handle crucial activities requiring for a quick response and accurate execution. The selection of programming language becomes critical to the success and efficacy of RTS systems as demand for them increases (Rammig, 2003).

Among the contemporary programming languages, Rust has been a strong competitor in recent years, attracting more interest and acceptance from software professionals across the globe. Since its 2010 release to the programming world, Rust—which was developed by Mozilla—has been well-known for its exceptional blend of efficiency, concurrency, and safety features. The language was designed with the aspirational mission of confronting the challenges involved in creating real-time applications while simultaneously offering a secure and effective alternative for conventional systems programming languages such as C and C++ (Bugden & Alahmar, 2022).

This literature review's goal is to examine the essential qualities of Rust which make it so highly suitable for the creation of real-time systems. This research aims to explain why Rust is the ideal language for developing real-time systems that require scalability, responsiveness, and safety through an examination at the language's concurrency primitives, performance optimisations, and safety implications.

This research aims to provide a deeper understanding of the appealing characteristics and benefits of Rust in the context of Real-Time Systems development by a thorough examination of current research findings, benchmarking research studies, and professional evaluations. Software engineers and researchers may choose the right programming language for their real-time projects by knowing the unique characteristics and capability of Rust. This will allow them to progress the field of real-time computing technology.

Unique Features of Rust

Rust's suitability for real-time applications is derived from several of unique characteristics that solve significant issues with robustness, memory management, and efficiency. Rust's ownership concept is a great example that fundamentally alters the way the language manages memory. Rust has an intricate ownership system that provides memory safety at compilation time without needing for a runtime garbage collector, in contrast to conventional programming languages that depend on garbage collection. Rust removes common memory-related bugs like use-after-free errors and double frees by keeping track of data ownership and complying with strict standards for memory allocation and deallocation (Obregon, 2024). That results in Rust especially suitable for real-time applications in which deterministic memory handling is crucial for reliable performance. In the context of replicating a real-time warehousing scenario for this assignment, Rust's ownership model assures precise and up-to- date information on inventory for dynamic stock replacement and optimisation, as well as immediate and reliable inventory tracking and supervision (Crichton, Gray, & Krishnamurthi, 2023).

Apart from its ownership model, Rust has solid concurrency support, that is crucial to effectively managing tasks that demand real-time processing. Concurrency primitives like threads and channels are incorporated into the language, which allows safe and effective concurrent programming. Rust's concurrency architecture imposes strict requirements for shared mutable state and offers synchronised communication channels between concurrent activities to avoid data races along with other concurrency-related issues. This guarantees that real-time systems can use parallelism efficiently while preserving the consistency and integrity of shared data, improving performance and reliability (Yu, Song, & Zhang, 2019). Hence, for this real-time warehousing simulation, Rust's concurrency primitives provide fluid coordination and management of goods transportation systems, making possible efficient movement of goods inside the warehouse environment.

Furthermore, memory safety and dependability in real-time applications are significantly improved by Rust's lifetime and borrowing mechanisms. The fundamental concepts of Rust's type system are lifetimes and borrowing, which allow a compiler to statistically analyse the lifetime of references and avoid common mistakes like data races and dangling pointers. Rust assures that a program's memory can be used securely and regularly all across the lifetime of a program by imposing stringent guidelines for reference lifetimes and data borrowing, lowering the risk of runtime errors and memory leaks. This degree of compile-time safety helps real-time systems, where errors might have serious consequences, be more dependable and predictable (Pearce, 2021). Rust's lifetimes and borrowing system protects the reliability of order processing systems in a real-time warehouse simulation by protecting against data corruption and providing precise validation and real-time processing of incoming orders.

Rust's approach to managing memory is set apart by its strict regulations and concentration on deterministic allocation of resources. Developers can create programmes that is both memory-efficient and free of typical memory-related issues thanks to the ownership architecture of the language and its support for lifetimes and borrowing. Rust's compiler ensures that resources are managed accurately and effectively during programme execution by enforcing strict regulations for memory allocation and deallocation. Rust is an ideal choice for building real-time systems where memory reliability and effectiveness are crucial because of its proactive approach to memory management, minimising the likelihood of memory corruption and other memory-related complications (Xuhui, ChenZhuangbin, SunMingshen, ZhouYangfan, & LyuMichael, 2021). Rust's memory management features ensure the smooth functioning of order processing, products transportation, and inventory management systems, allowing reliable as well as efficient real-time warehouse operations.

To sum up, Rust is an excellent choice for Real-Time Systems because of its ownership model, concurrency primitives, lifetimes and borrowing, and memory management approach, among other particular features. This is especially true when emulating sophisticated warehousing scenarios. Through the provision of memory safety, concurrency support, and deterministic resource management, Rust allows developers to develop robust and effective real-time applications, hence facilitating the development of real-time computing technology across diverse sectors.

## Concerns Addressed by Rust

Rust improves the efficiency, robustness, and reliability of real-time systems by addressing a variety of issues that occur in these systems. Let's explore each aspect.

In real-time systems, where the smallest errors can have serious repercussions, reliability is vital. Rust takes a proactive stance, preventing common pitfalls like null pointer dereferences and data races by performing error checks during the compilation process. Rust enhances the framework of real-time systems by recognising these problems promptly throughout the development process, ensuring that they operate smoothly and predictably (Balasubramanian, et al., 2017).

Robustness is also important because real-time systems have to deal with errors and unforeseen events without sacrificing performance. a solid error management becomes achievable by Rust's Result and Option types. Rust employs mechanisms such as Result and Option types to provide users with robust error management. By encouraging developers to address errors explicitly, these constructs allow them to become more resilient in the face of unforeseen difficulties (Li, Ding, Lee, Sha, & Caccamo, 2006). Rust provides real-time systems with the capability to gracefully navigate challenging environments while maintaining their reliability and functionality by encouraging methodical and explicit error handling techniques.

Regarding real-time systems, which frequently function under strict time and resource constraints, efficiency is a crucial factor. Rust is outstanding in this sector by providing performance optimisations that positioned efficiency ahead of safety. Rust gives developers the ability to develop high-performance code that minimises overhead and maximises resource utilisation through features like precise memory management and zero-cost abstractions. Rust-based real-time systems can achieve higher levels of responsiveness and efficiency by utilising these features, even in demanding environment (Stoyenko, 1992).

In summary, Rust is a compelling option for creating real-time systems due to its emphasis on robustness, efficiency, and reliability. Through proactive mitigation of common pitfalls, solid error management, and performance optimisation, Rust enables developers to build real-time applications that satisfy the demanding needs of an extensive variety of applications and industries.

## Linking Concepts in Rust

The manner in which Rust's features and principles are interconnected greatly influences how programmes are designed and operate and Rust's lifetime and ownership mechanisms serve as the foundation for safe concurrency. Rust reduces common pitfalls like data races by enforcing strict regulations around memory management and accessibility, ensuring that multiple threads can run concurrently without compromising the reliability of shared resources (Tripathi, Yadav, Ranvijay, & Jana, 2011). By minimising the risk of irregular behaviour, this not only strengthens the integrity of real-time systems but also boosts their efficiency by permitting task execution in parallel.

In addition, Rust's memory management model is crucial for ensuring predictable resource consumption and reducing runtime overhead—two things that are necessary to satisfy the demanding real-time system performance standards. Rust allows developers to optimise memory utilisation and bypass expensive runtime procedures like garbage collection by employing concepts like stack allocation and deterministic deallocation by carrying out ownership and borrowing (Chen, 2014). This enhances the reactivity and scaling of applications by facilitating the execution of real-time tasks more smoothly and making better use of available resources.

Additionally, Rust's concurrency primitives and thread model enable its support for priority-based scheduling mechanisms, providing developers with an effective tool for regulating task execution in real-time systems. Rust enables developers prioritise and perform critical operations precisely by giving tasks a priority and managing their execution according to pre-established scheduling policies (Cedeño & Laplante, 2007). With this feature, developers are able to satisfy strict time constraints and remain responsive in real-time applications, especially when tasks have different degrees of urgency or importance.

In conclusion, Rust's concepts and features are closely related to one another and reinforce each other, making it possible for programmers to create real-time systems that are flexible enough to adapt to shifting priorities and demands while remaining dependable and efficient. Rust enables developers to develop real-time programmes that excel in performance and reliability by utilising ownership and lifetimes for secure concurrency, optimising memory management for predictable consumption of resources, and putting priority scheduling mechanisms in place for task coordination.

## Conclusion

In conclusion, Rust emerges as a standout choice for developing Real-Time Systems, thanks to its unique features and robust capabilities. Through our exploration, we've witnessed how Rust's ownership, concurrency, and memory management models synergize to ensure safety, reliability, and efficiency in real-time applications. By addressing key concerns such as reliability, robustness, and efficiency, Rust paves the way for innovative advancements in real-time computing. As we look ahead, Rust's potential to drive innovation and shape the future of real-time application development remains unparalleled, making it a compelling option for developers seeking to build cutting-edge solutions in this dynamic domain.

# 4.0 Design and Methodology

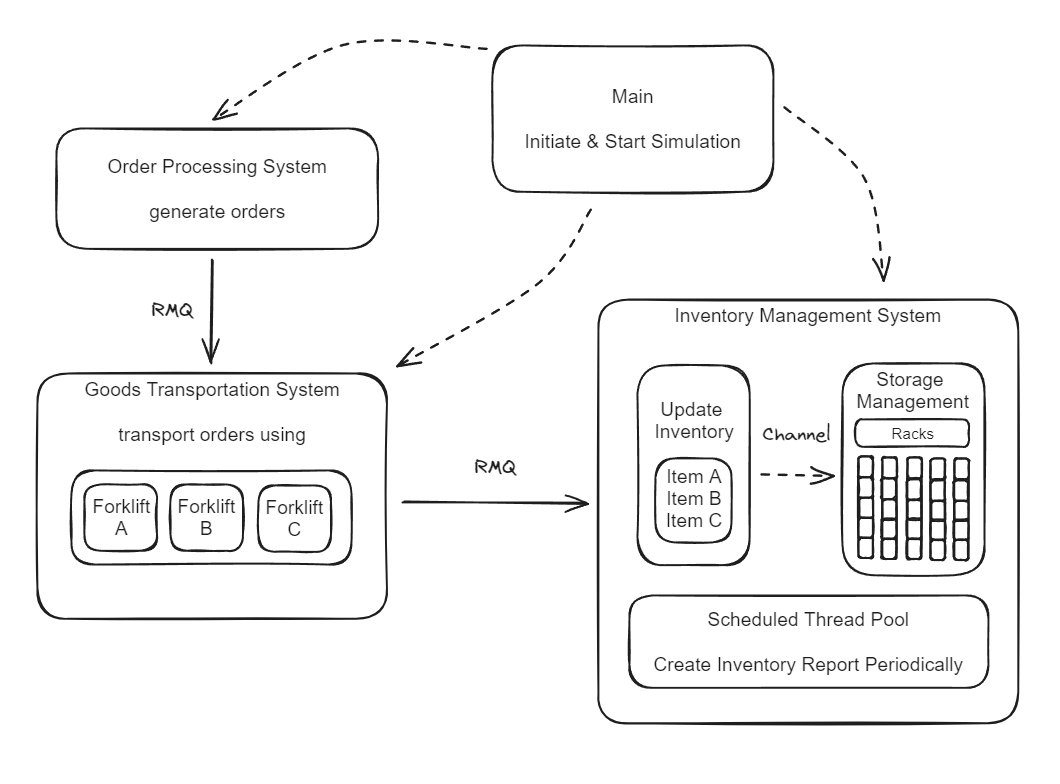


Figure 1 System Design

This project involves designing a multi-threaded simulation of a simple warehouse management system (WMS). The goal is to create a realistic scenario of warehouse operations with real-time communication and data processing.

The simulation begins with the order processing system, which generates random orders. These orders can either be supplies arriving from outside sources or requests to offload items from the warehouse. The system randomly determines the quantity, frequency of orders, and items to simulate real-life warehouse activities. Before processing offload orders, the system checks the availability of the stock to ensure that there is sufficient inventory.

A RabbitMQ communication service is established between the order processing system and the goods transportation system. This enables the seamless transfer of orders. In this simulation, forklifts represent the transportation of goods, using mutexes to ensure shared resources are managed correctly and to prevent deadlocks.

Another RabbitMQ communication link connects the goods transportation system and the inventory management system (IMS). When goods are transported, the IMS is updated in real-time to reflect the changes in inventory. The IMS also includes a simulation of a storage area with racks to store items. Communication between the inventory update and storage management is handled using channels, ensuring that orders and items are synchronized in real-time. This real-time synchronization prevents errors and allows other systems to access up-to-date information.

Mutexes (short for mutual exclusions) are critical in this simulation to handle concurrency issues. In a multi-threaded environment, multiple threads may attempt to access or modify the same data simultaneously. Without proper synchronization, this can lead to race conditions, where the outcome depends on the sequence of thread execution. By using mutexes, we ensure that only one thread can access a resource at a time, thus preventing data corruption and ensuring consistency. This is especially important in real-time systems where timely and accurate data processing is crucial.

To simulate periodic inventory reporting, a scheduled thread pool generates inventory reports at regular intervals. This mimic the actions of a warehouse manager who periodically checks the status of the inventory to ensure everything is in order.

To summarize, this project creates a dynamic, multi-threaded simulation of a warehouse management system, focusing on real-time data processing and communication. By leveraging Rust's concurrency features and using RabbitMQ for message passing, we ensure that the system operates smoothly and efficiently, reflecting real-world warehouse operations. Mutexes play a vital role in managing concurrent access to shared resources, preventing data inconsistencies and ensuring reliable system behaviour. Periodic inventory reporting adds an extra layer of realism, akin to a warehouse manager's regular checks. This detailed and robust simulation showcases the power of Rust in building safe and efficient real-time systems.

## Coding Implementation and Explanation

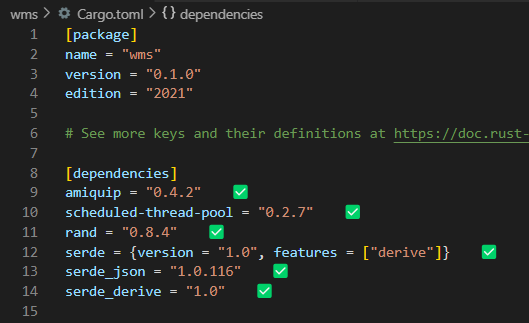


Figure 2 Cargo.toml

This is my Cargo.toml screenshot, it consist of the package details and my dependencies used for this project, such as amiquip is for setting up the Rabbit MQ message broker, and serde for serialization and deserialization of data.

A screenshot of a computer

Description automatically generated

Figure 3 Project Directory

The package name is “wms” which stands for Warehouse Management System. There are five rust files in the source directory.

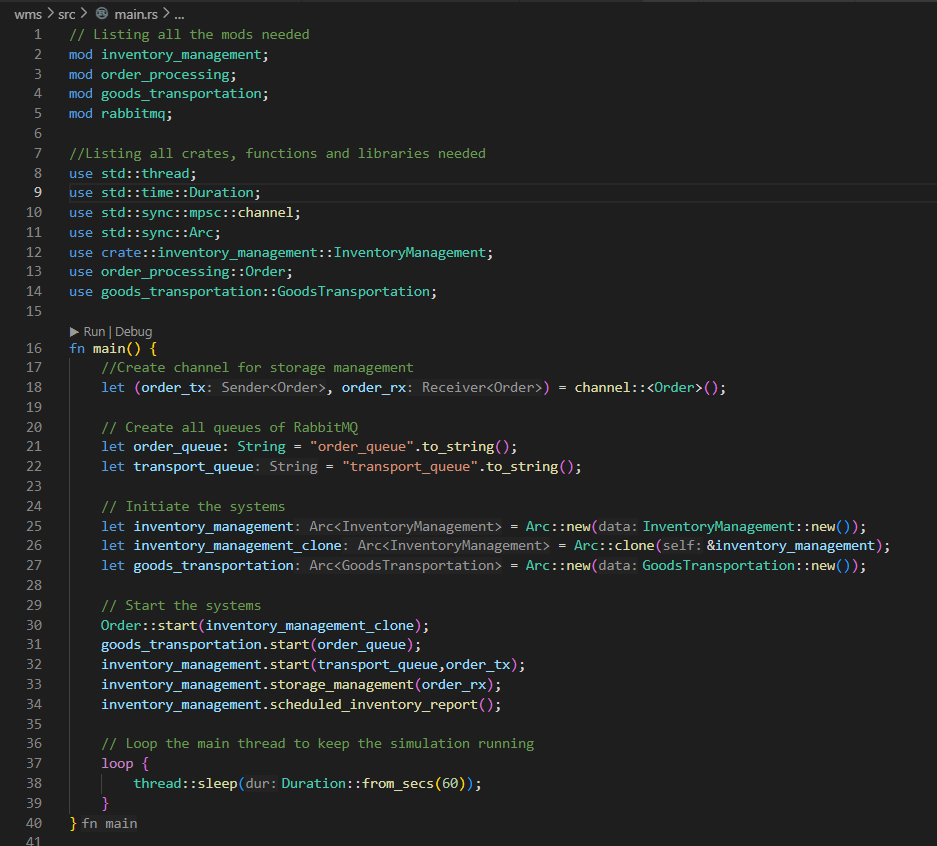
\

Figure 4 Main.rs – Mods and Imports

In this section, the necessary modules and crates are imported and declared. The custom modules include inventory\_management, order\_processing, goods\_transportation, and rabbitmq. The std::thread and std::time::Duration are used for thread handling and sleep intervals, respectively. The std::sync::mpsc::channel provides a way to create channels for message passing, and std::sync::Arc is used for thread-safe reference counting.

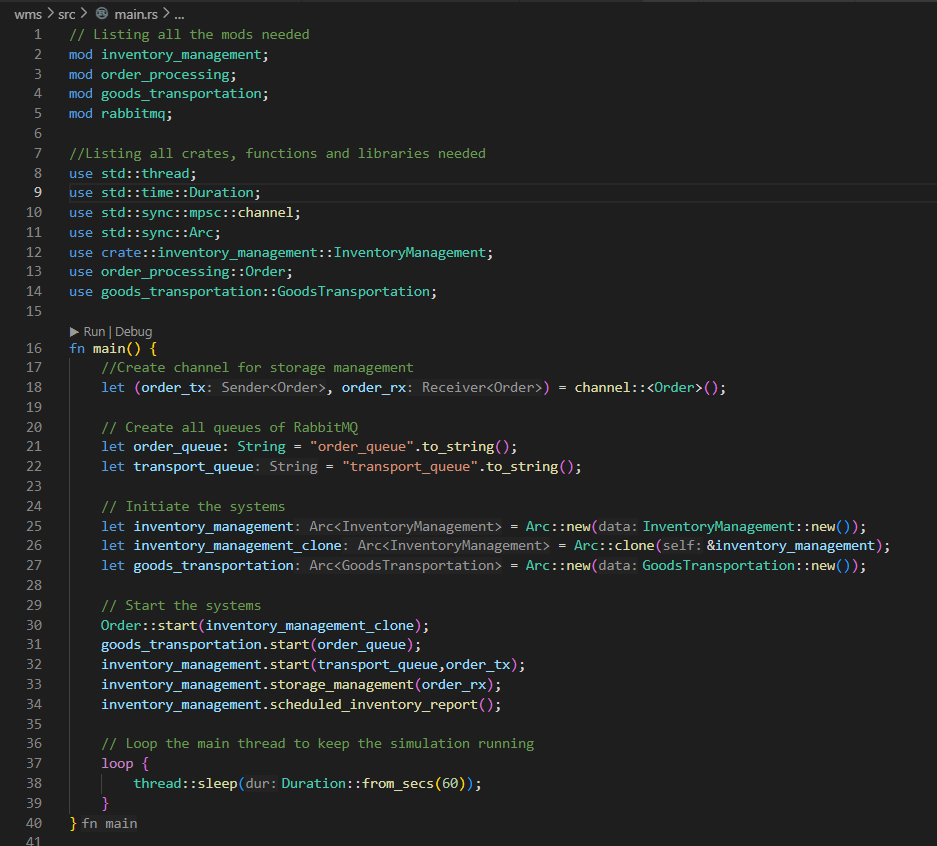


Figure 5 Main.rs - System Simulation Start

The main.rs file is the entry point of the warehouse management system simulation. It sets up inter-module communication using channels and RabbitMQ, initializes key components (order processing, goods transportation, and inventory management), and starts these systems to work concurrently. The Arc and Mutex ensure thread-safe operations, enabling real-time updates and synchronization across the system. The scheduled inventory report feature adds an element of periodic checks, reflecting realistic warehouse management practices. This setup demonstrates the effective use of concurrency and real-time data handling in Rust.

A screen shot of a computer program

Description automatically generated

Figure 6 order\_processing.rs – Imports and Order Struct

This section imports the necessary modules and crates for the order processing system. The std::thread and std::sync::Arc modules handle threading and shared references, respectively. The rand crate generates random numbers to simulate real-life order variability. The custom inventory\_management and rabbitmq modules provide functions for inventory management and message sending. The serde and serde\_json crates are used for serializing and deserializing data. It also defined Order struct, which represents an order in the system. Each order includes an index, item code, quantity, and order type. The struct is derived with Debug, Clone, Serialize, and Deserialize traits to facilitate debugging, cloning, and JSON serialization/deserialization.

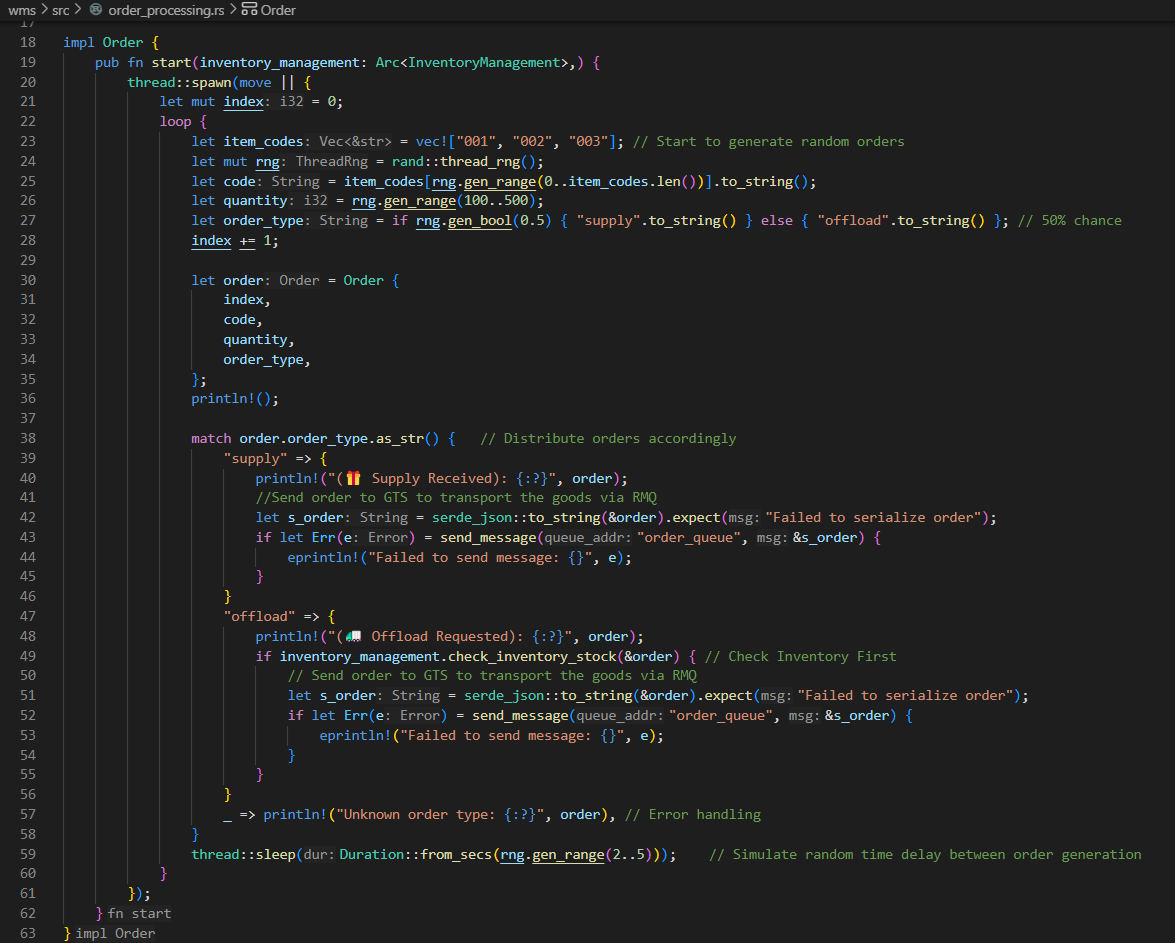


Figure 7 order\_processing.rs – Order Implementation Part 1

The Order struct has a start function that kicks off a new thread to generate and process orders. This is done using thread::spawn, which allows the main simulation to continue running concurrently. Within the thread, a loop continuously generates random orders. Each order is assigned an index, item code (randomly selected from a predefined list), quantity, and order type (randomly chosen as either "supply" or "offload" with equal probability). This randomness simulates real-life order variability.

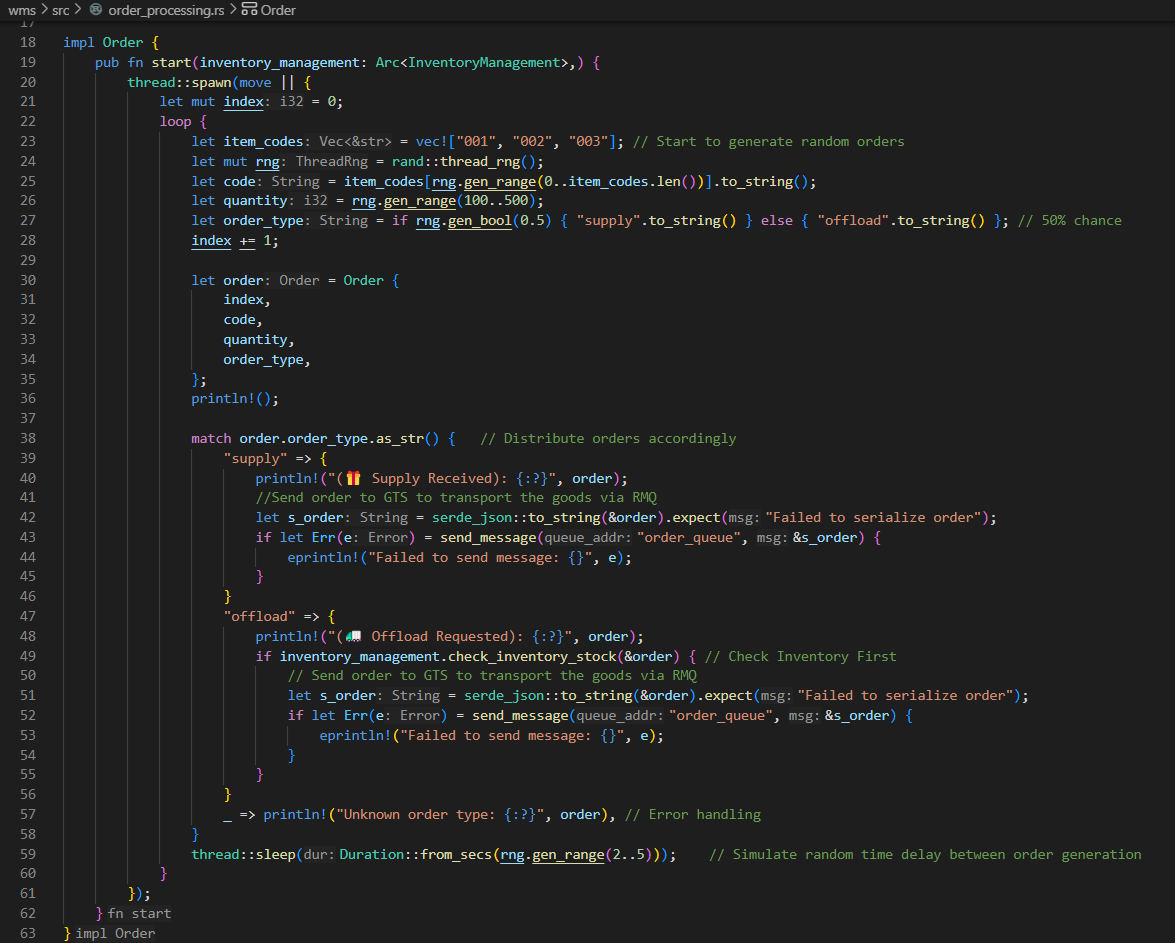


Figure 8 order\_processing.rs – Order Implementation Part 2

Each generated order is then printed to the console for logging purposes. If the order is a "supply" type, it is serialized to JSON and sent to the "order\_queue" using RabbitMQ. If the order is an "offload" type, the system first checks if there is enough stock available using the check\_inventory\_stock method from the InventoryManagement module. If there is enough stock, the order is serialized and sent to the "order\_queue". To simulate a realistic delay between order generations, the thread sleeps for a random duration between 2 and 5 seconds before generating the next order. This adds a layer of realism to the simulation, mimicking the natural arrival of orders over time.

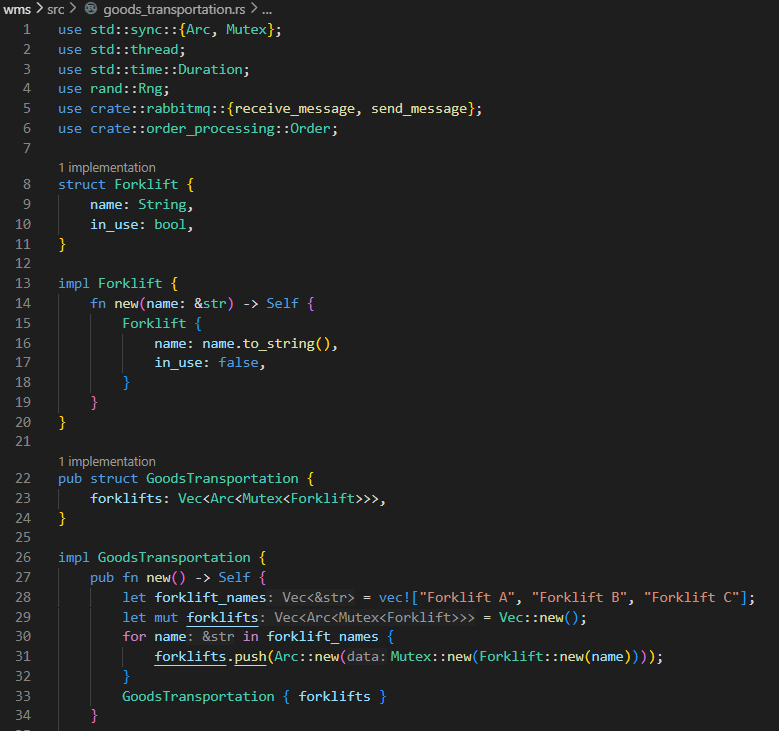


Figure 9 goods\_transportation.rs – Imports, Forklift Struct, Impl of GoodsTransportation

This section imports essential modules and crates necessary for the goods transportation system. The std::sync::{Arc, Mutex} modules are used for managing shared resources safely across threads. The rand crate generates random numbers to simulate real-life delays. Custom modules rabbitmq and order\_processing handle messaging and order data structures, respectively. It also defines the Forklift struct, representing each forklift in the system. Each forklift has a name and an in\_use flag to indicate whether it is currently busy. The new method initializes a forklift with a given name, setting in\_use to false by default. The GoodsTransportation struct manages multiple forklifts. In the new method, it initializes a list of forklifts with predefined names. Each forklift is wrapped in Arc<Mutex<>> to allow safe shared access across threads.

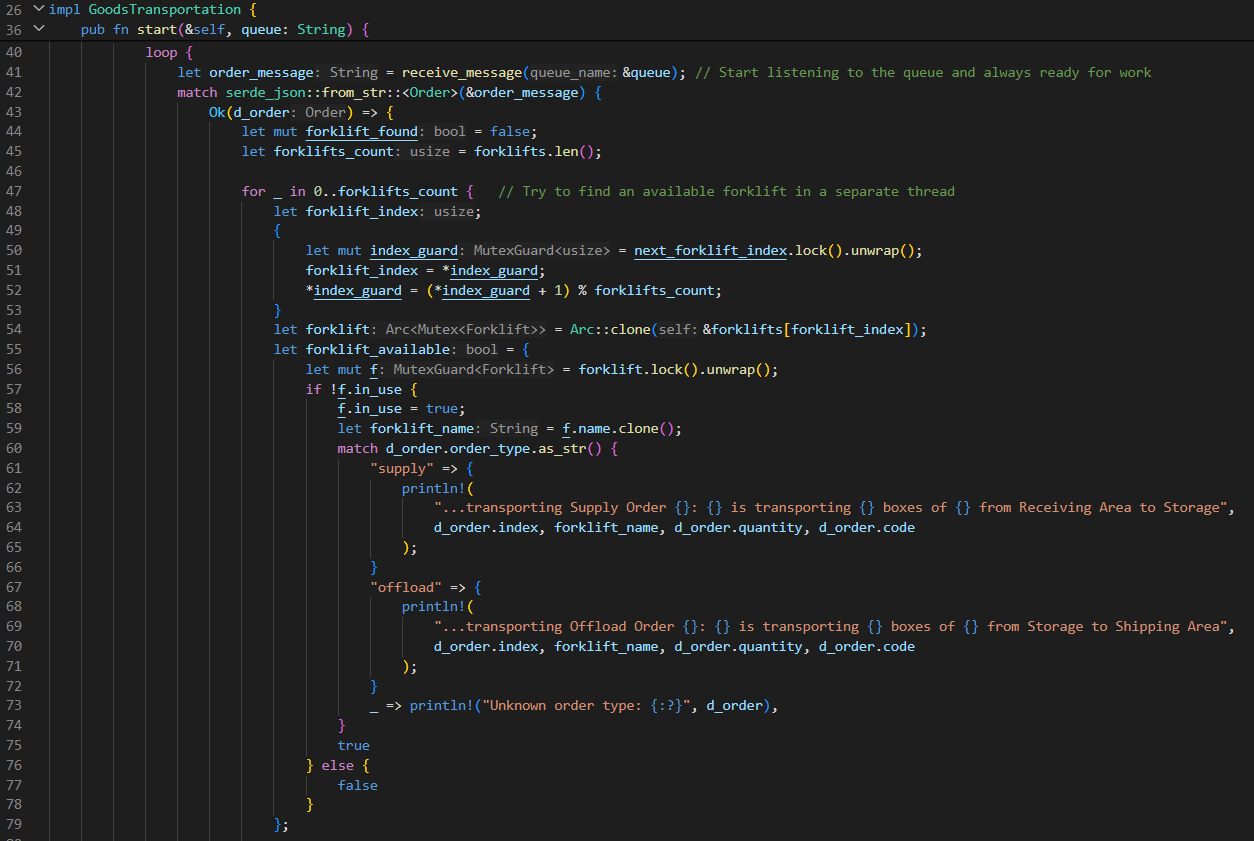


Figure 10 goods\_transportation.rs – Start function Part 1

The start method initiates a new thread to handle incoming orders continuously. It listens to a queue for order messages, deserializes each message into an Order struct, and tries to find an available forklift to handle the order. To distribute work evenly among forklifts, a round-robin algorithm is used. The next\_forklift\_index tracks the next forklift to be assigned. This index is updated in a cyclic manner to ensure all forklifts get a fair share of the workload. This method prevents any single forklift from being overburdened while others remain idle, optimizing resource utilization.

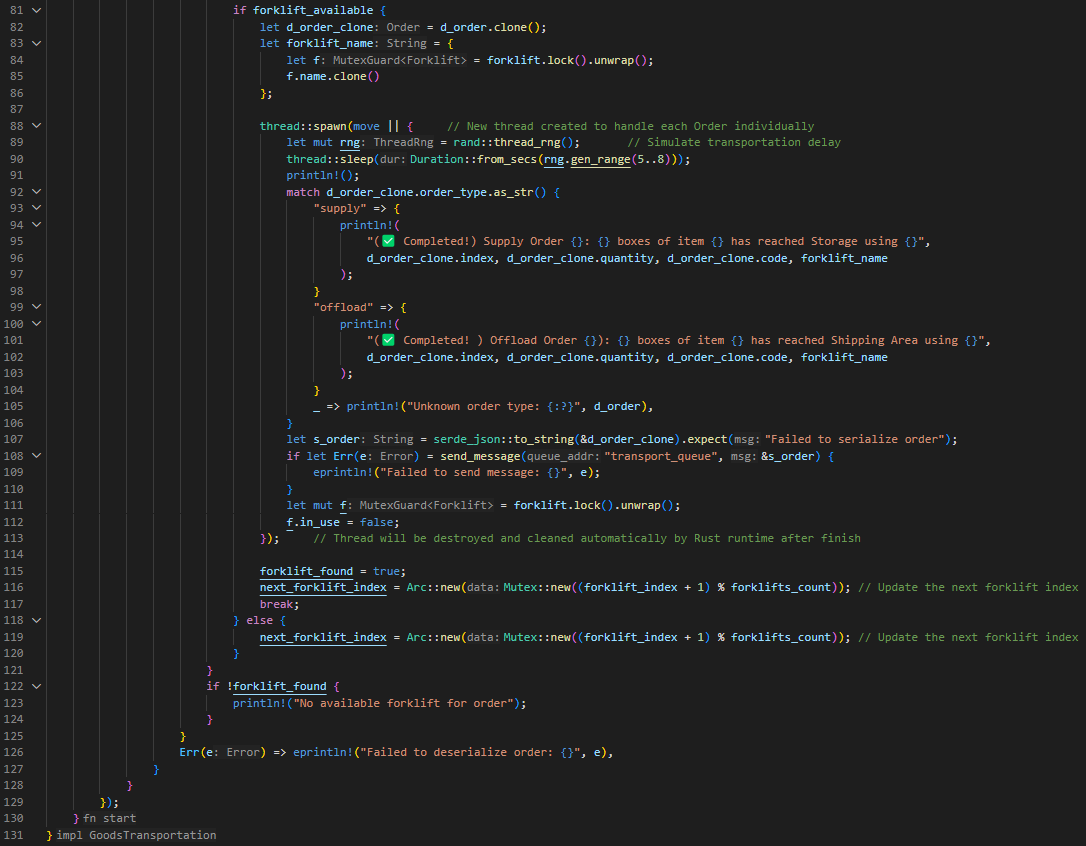


Figure 11 goods\_transportation.rs – Start function Part 2

Continue on from the same function, the method iterates through the forklifts to find one that is not in use. If an available forklift is found, it locks the forklift for use, prints a message indicating the transportation of the order, and starts a new thread to simulate the transportation process, after the thread completed its tasks, it will be destroyed and cleared automatically by Rust runtime. Once the transportation is completed, it sends a message to update the system and releases the forklift for future use. Each order's transportation is simulated by putting the thread to sleep for a random duration, mimicking real-life delays. After the delay, a completion message is printed, and the order is marked as completed and sent to the next stage via RabbitMQ. In summary, this code sets up a goods transportation system that efficiently manages forklifts to handle orders using a round-robin method. This approach ensures even distribution of tasks, prevents overburdening any single forklift, and optimizes resource use. The system operates in real-time, processing and updating orders dynamically through multithreading and messaging.

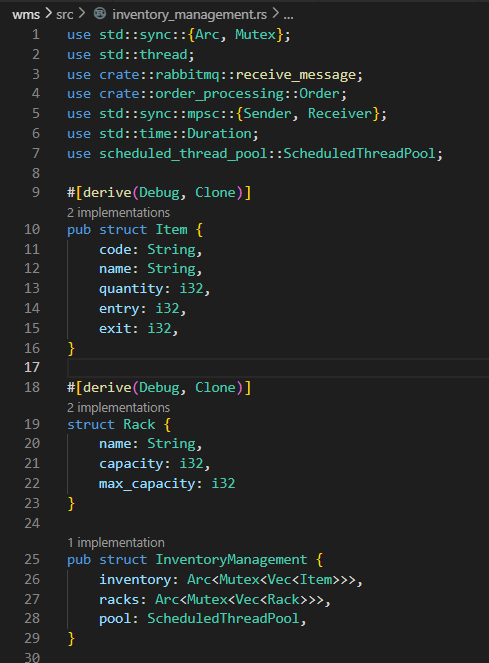


Figure 12 inventory\_management.rs – Imports, Item Struct & Rack Struct

This section imports essential libraries and modules. Arc and Mutex manage shared resources safely across threads, thread handles multithreading, and mpsc facilitates message passing between threads. ScheduledThreadPool is used for scheduling tasks at fixed intervals. Custom modules rabbitmq and order\_processing handle messaging and order data structures. The Item struct represents items in the inventory, including their code, name, quantity, and entry/exit counts. The Rack struct represents storage racks in the warehouse, with their name, current capacity, and maximum capacity.



Figure 13 inventory\_management.rs – Impl of InventoryManagement, new()

The InventoryManagement struct manages the inventory and racks. The new method initializes the inventory with a list of items and the racks with their initial capacities. Both inventory and racks are wrapped in Arc<Mutex<>> for safe shared access across threads. A thread pool is also created for scheduled tasks.

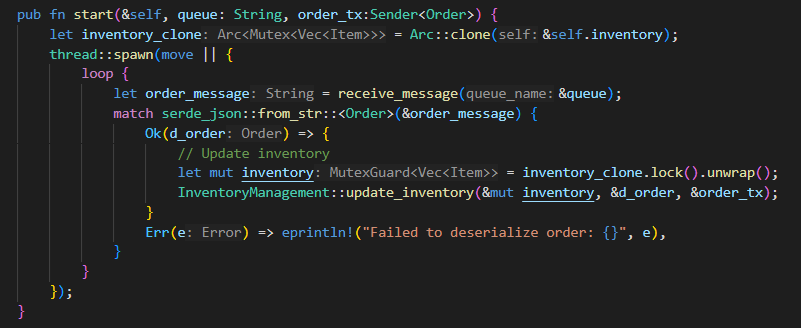


Figure 14 inventory\_management.rs – start()

The start method creates a new thread to continuously listen for incoming order messages from a queue. When an order is received, it deserializes the message into an Order struct and updates the inventory accordingly by calling update\_inventory.

A screen shot of a computer code

Description automatically generated

Figure 15 inventory\_management.rs – update\_inventory()

The update\_inventory method adjusts the inventory based on the order type. For a supply order, it increases the item’s entry count and quantity. For an offload order, it increases the exit count and decreases the quantity. The updated order is then sent through the provided channel.

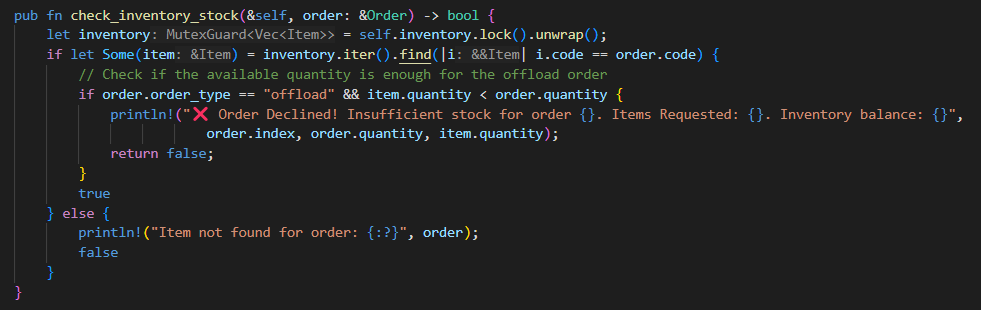


Figure 16 inventory\_management.rs – check\_inventory\_stock()

The check\_inventory\_stock method checks if there is enough stock available for an offload order. If the stock is insufficient, it prints a message and returns false. Otherwise, it returns true, indicating that the order can proceed.



Figure 17 inventory\_management.rs – storage\_management()

The storage\_management method processes orders to update the storage racks. For supply orders, it distributes the incoming items across racks based on available capacity. For offload orders, it removes items from the racks accordingly. This ensures that the storage system remains organized and up-to-date.

A screen shot of a computer code

Description automatically generated

Figure 18 inventory\_management.rs – scheduled\_inventory\_report()

The scheduled\_inventory\_report method sets up a periodic task to generate and print inventory and rack status reports. Using a thread pool, it schedules this task to run at fixed intervals, ensuring regular updates on the state of the inventory and storage racks. This helps in monitoring and maintaining the warehouse efficiently.

A screen shot of a computer program

Description automatically generated

Figure 19 rabbitmq.rs – Rabbit MQ code, send\_message()

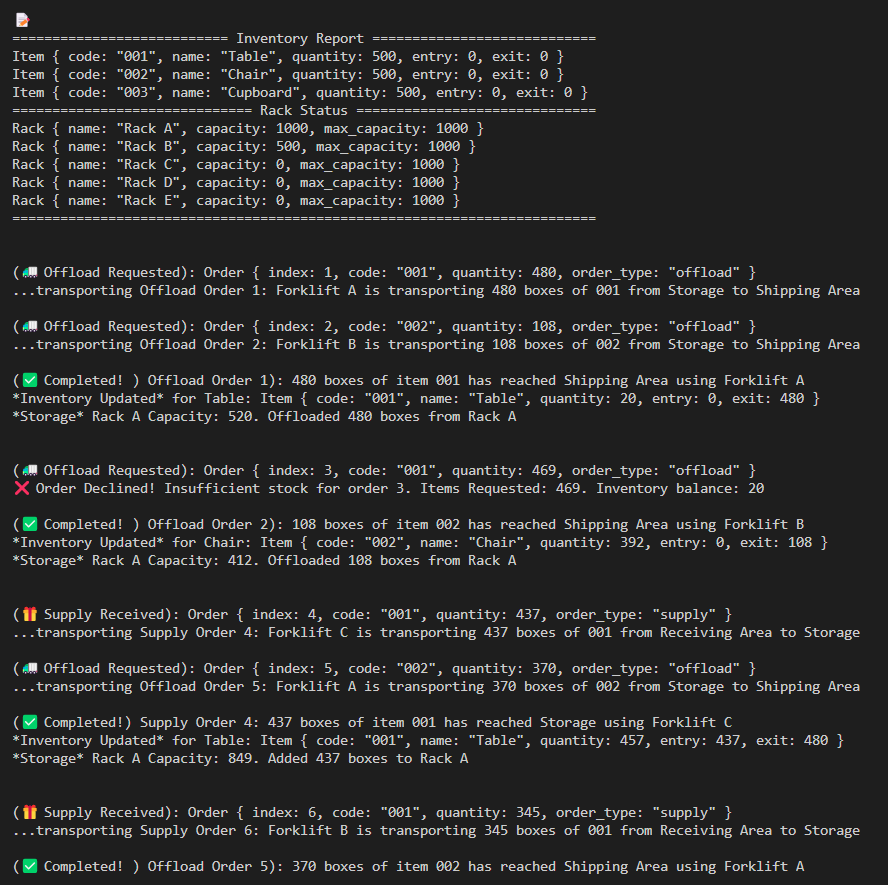
The rabbitmq.rs imports necessary modules from the amiquip library, which is used to interact with RabbitMQ for sending and receiving messages. Connection, Exchange, and Publish are for establishing connections and publishing messages, while ConsumerMessage, ConsumerOptions, and QueueDeclareOptions are for handling message consumption. The send\_message function is responsible for publishing messages to a specified queue in RabbitMQ. It starts by establishing an insecure connection to the RabbitMQ server using default credentials. A channel is then opened, and a direct exchange is created. The message is published to the specified queue address, and the connection is closed afterward. The function returns a Result to indicate success or failure.

A screen shot of a computer program

Description automatically generated

Figure 20 rabbitmq.rs – Rabbit MQ code, receive\_message()

The receive\_message function listens for messages from a specified RabbitMQ queue and retrieves them. It opens an insecure connection to the RabbitMQ server and establishes a channel. The queue is declared, and a consumer is set up to listen for messages. When a message is received, it is converted to a string, acknowledged, and stored in the msg variable. If the consumer ends for any reason other than receiving a message, an appropriate message is printed. The connection is then closed, and the message is returned. In summary, these two functions handle communication with a RabbitMQ server. send\_message publishes a message to a specified queue, establishing a connection, opening a channel, and using a direct exchange. receive\_message listens for and retrieves messages from a specified queue, converting them to a string and acknowledging receipt. Both functions ensure the connection is closed after operations are complete, maintaining proper resource management.



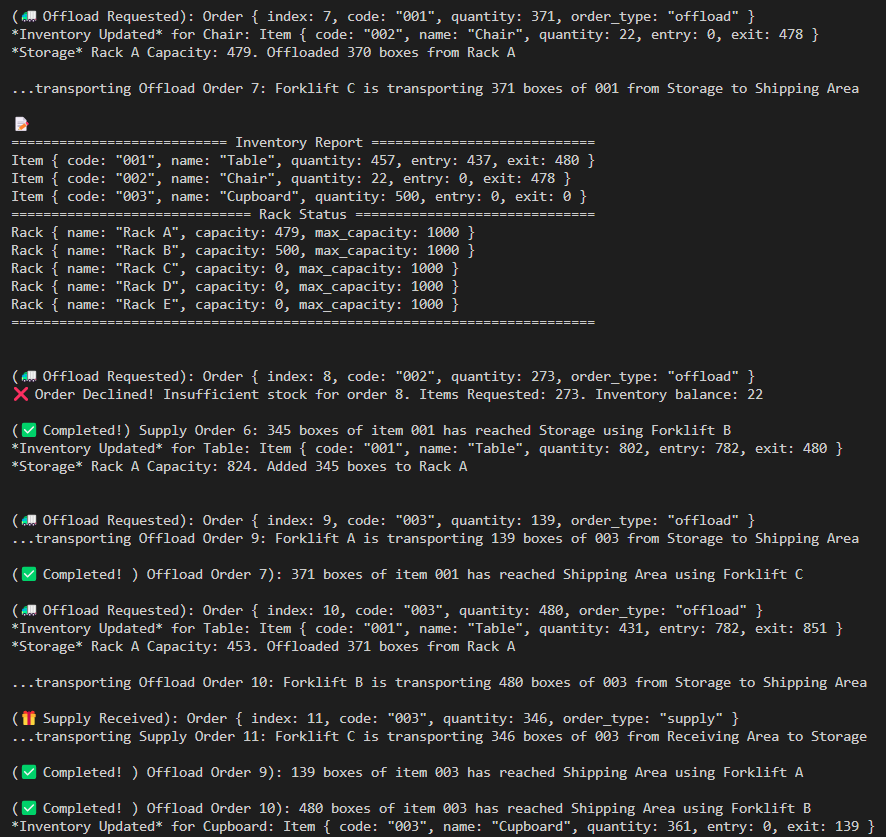


Figure 21 Code Output

The initial inventory report shows that each item started with a quantity of 500, and no items had been moved in or out at the start. The racks were initially set up with varying capacities, where Rack A: 1000 (full capacity), Rack B: 500 and Rack C, D, E: Empty. This setup provides a clear starting point for monitoring changes as orders are processed.

**Order Processing and Inventory Updates**

**Order Fulfillment and Inventory Adjustments:**

The system successfully handles offload and supply requests, updating inventory levels accurately. For example:

* Order 1 offloaded 480 tables (code: 001), reducing the quantity from 500 to 20.
* Order 2 offloaded 108 chairs (code: 002), reducing the quantity from 500 to 392.
* Supply orders were also handled effectively, such as Order 4 adding 437 tables, raising the quantity from 20 to 457.

**Error Handling:**

This robust error handling ensures that the system maintains inventory integrity and prevents negative stock levels. The system correctly identifies and declines orders when there is insufficient stock. For instance:

* Order 3 requested 469 tables but was declined as only 20 were available.
* Order 8 and Order 13 for chairs (code: 002) were declined due to insufficient stock.

**Capacity Management:**

Rack capacities are dynamically updated based on the transactions. This effective capacity management indicates the system’s ability to distribute items across multiple storage areas efficiently. For example:

* After fulfilling Order 1, Rack A's capacity increased to 520 from 1000.
* Supply Order 18 added 438 chairs, maxing out Rack A's capacity and distributing the surplus to Rack B and Rack C.

**Detailed Observations**

The system maintains consistent inventory levels and rack capacities. Each transaction is followed by an accurate update of the relevant metrics, ensuring data reliability. There are no signs of data races or concurrency issues. Transactions are processed sequentially, and inventory updates reflect the latest state after each transaction. Error messages are clear and informative, providing specific reasons for order declines. For example: “Order Declined! Insufficient stock for order 3. Items Requested: 469. Inventory balance: 20." This level of detail in logging helps in diagnosing issues quickly and maintaining operational transparency.

**Conclusion**

The system demonstrates robust functionality in handling inventory and orders, with effective error handling, detailed logging, and dynamic capacity management. There are no significant concurrency issues, and the few minor anomalies observed do not detract significantly from the system's overall reliability. Addressing the graceful termination and ensuring atomic transactions for edge cases will enhance the system’s robustness further. This thorough and systematic benchmarking indicates a well-designed inventory management system ready for deployment with minor adjustments.

# 5.0 Result and Discussion

## Disclaimer

While this project was intended as a collaborative effort of two members, due to unforeseen circumstances, my teammate was unable to complete his portion of the work. As a result, the benchmarking and comparison aspect of the assignment could not be fulfilled. It is important to acknowledge that collaboration fosters diverse perspectives and insights, contributing to a richer learning experience. Despite the absence of comparative analysis, this documentation presents my individual efforts in developing and benchmarking the warehouse management system.

## 5.1 Benchmark 1 – Order Generation

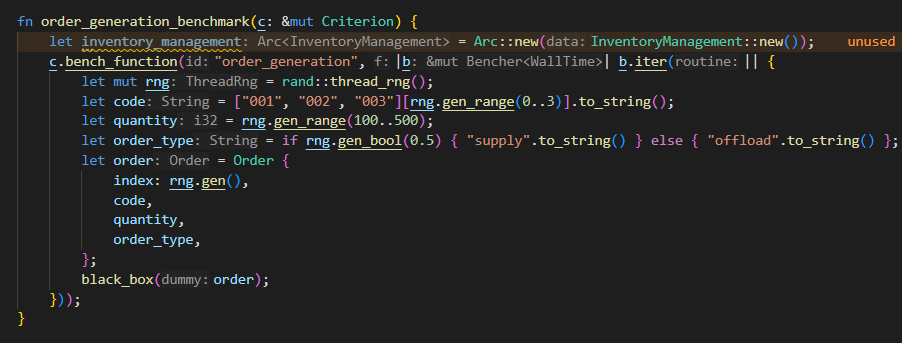


Figure 22 Benchmark 1 – Function

This function simulates random order generation to kickstart the system. It randomly generates index number, order code, order quantity, order type for an order.

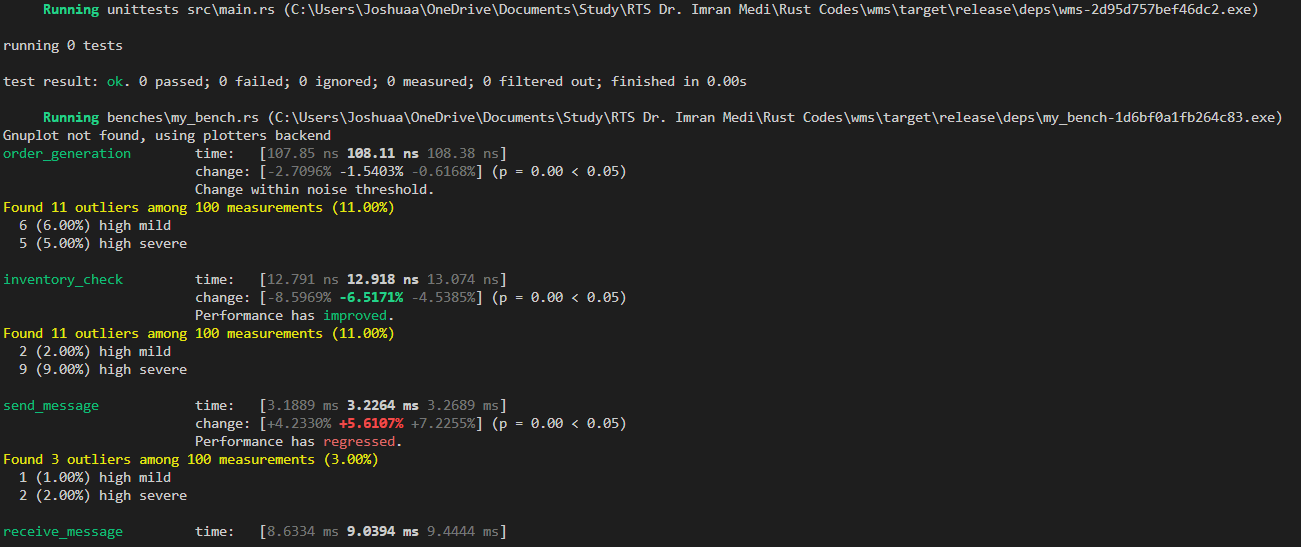


Figure 23 Benchmark 1 - Terminal

The “order\_generation” function has an average execution time of 107.85ns. Out of 100 measurements, there are 11 outliers (11%) found, including 6 high mild and 5 high severe.

A graph and chart with text

Description automatically generated

Figure 24 Benchmark 1 – Criterion Report

The report above stated that for each additional run, the time taken increases by around 107.85 nanoseconds on average. The R² value, which shows how well the data fits a straight line, is high at about 0.986, suggesting a strong connection between the number of runs and total time. The average time for each run is around 107.65 ns, with half of the runs taking less than 107.34 ns. The mean absolute deviation (MAD), a measure of how much the times deviate from the average, is about 351.91 picoseconds, and the standard deviation, showing how spread out the times are, is around 819.37 ps. Overall, it looks like there's a consistent pattern of increasing time with more runs, but it's within a certain range, which is decent for performance.

## 5.2 Benchmark 2 – Send Message

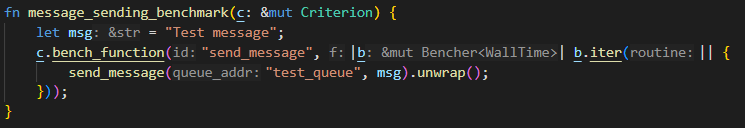


Figure 25 Benchmark 2 – Function

This “send\_message” function is a part of the Rabbit MQ communication setup, where it provide the function for system to send message into a particular queue.

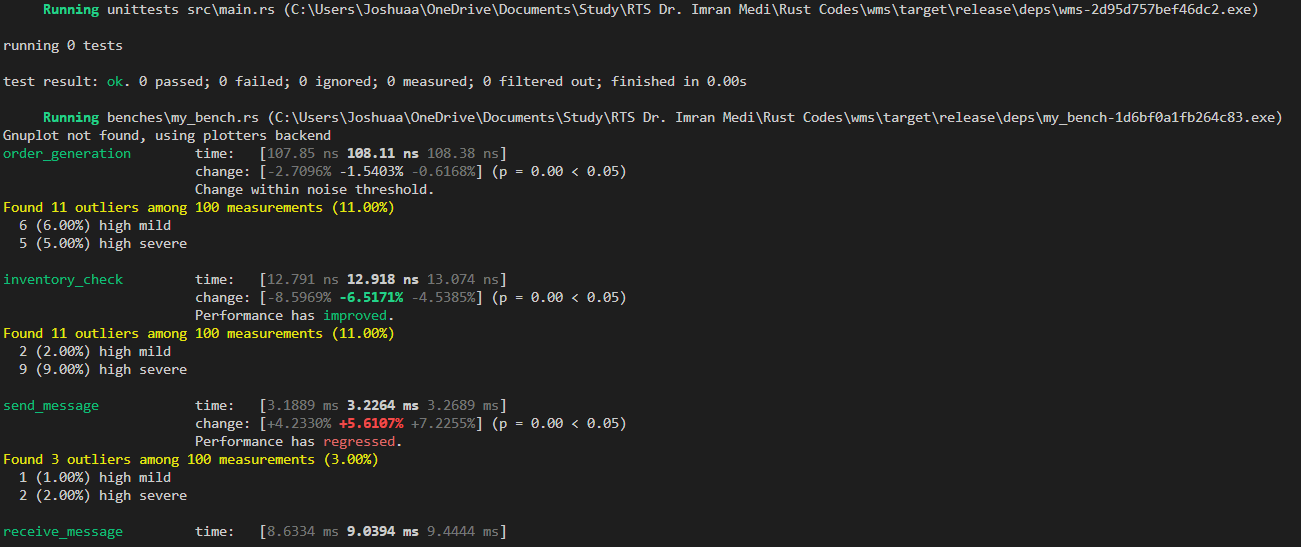


Figure 26 Benchmark 2 - Terminal

The “send\_message” function has an average execution time of 3.2265 ms. Out of 100 measurements, there are 3 outliers (3%) found, including 1 high mild and 2 high severe.

A graph and diagram of a graph

Description automatically generated with medium confidence

Figure 27 Benchmark 2 – Criterion Report

In the case of the "send\_message" function, there's no significant trend in execution time with each additional run, as indicated by the very low R² value of approximately 0.0027029. This suggests a weak correlation between the number of runs and the total execution time. The mean average time for each run is around 3.2264 milliseconds, with the median slightly lower at 3.2181 milliseconds. The mean absolute deviation (MAD) is relatively low at about 139.37 microseconds, indicating limited deviation from the average time, while the standard deviation is around 206.36 microseconds, showing moderate dispersion of times around the mean. Overall, these metrics suggest that the execution time for the "send\_message" function varies within a certain range without a clear pattern of increase or decrease with more runs, indicating moderate stability in performance.

## 5.3 Benchmark 3 – Receive Message

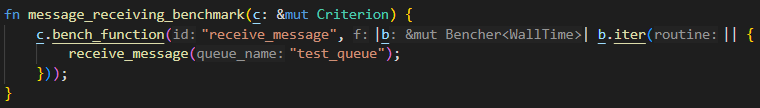


Figure 28 Benchmark 3 - Function

This “receive\_message” function is the other part of the Rabbit MQ communication message broker setup, where it allows processes to receive message from the queue, but there need to be message sent.

A screen shot of a computer program

Description automatically generated

Figure 29 Benchmark 3 – Terminal

In the terminal, the “receive\_message” function have an average execution time of 9.0394 ms.

A diagram and graph with text

Description automatically generated with medium confidence

Figure 30 Benchmark 3 – Criterion Report

For the "receive\_message" function, the R² value of approximately 0.0097977 suggests a weak correlation between the number of runs and the total execution time, indicating no discernible trend in performance with increasing iterations. The mean average time for each run is around 9.0394 milliseconds, with a slightly higher median at 9.2154 milliseconds. The mean absolute deviation (MAD) is approximately 3.1063 milliseconds, indicating some deviation from the average time, while the standard deviation is around 2.0666 milliseconds, suggesting moderate dispersion of times around the mean. Overall, these metrics indicate that the execution time for the "receive\_message" function varies within a certain range without a clear pattern of increase or decrease with more runs, highlighting moderate stability in performance.

## 5.4 Benchmark 4 – Inventory Check

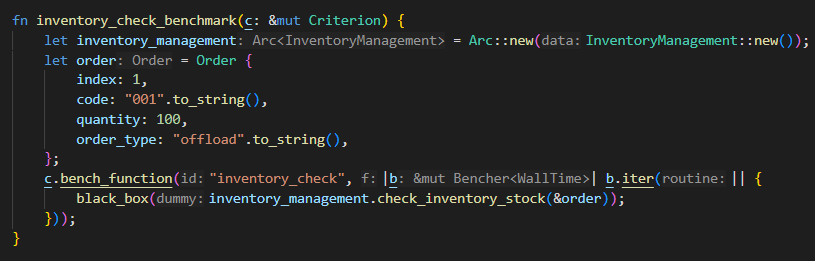


Figure 31 Benchmark 4 – Function

This “inventory\_check” function takes in an order struct, with index, order code, order quantity and order type, then it checks in the item vectors, whether there are enough stock for offload orders to be made.

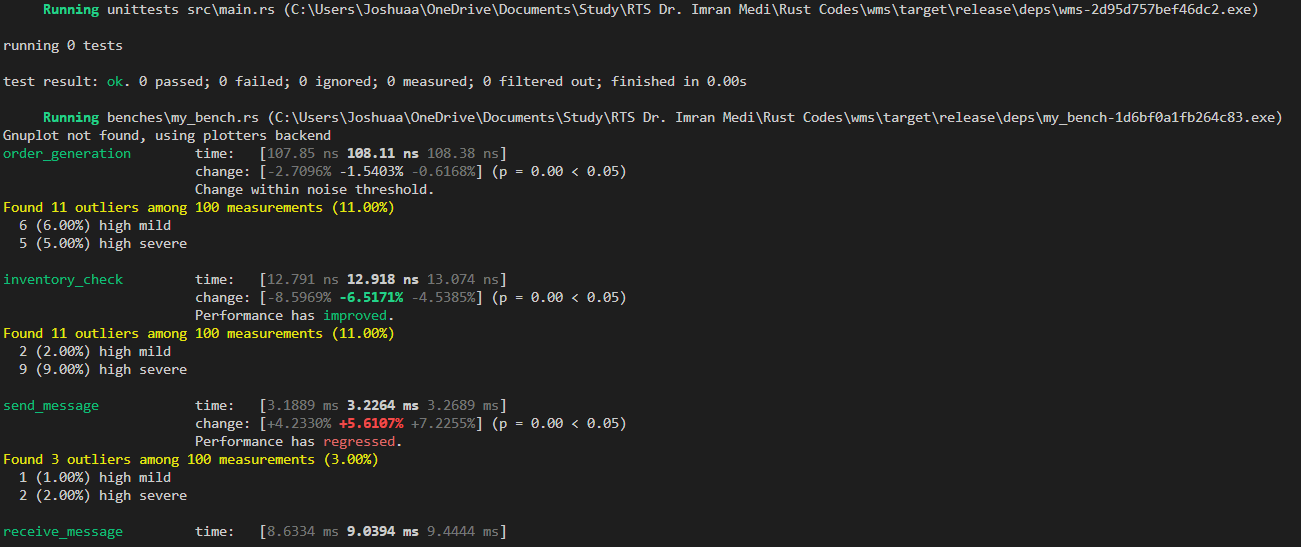


Figure 32 Benchmark 4 - Terminal

In the terminal, the “inventory\_check” function has an average execution time of 12.918ns. Out of 100 measurements, there are 11 outliers (11%) found, including 2 high mild and 9 high severe.

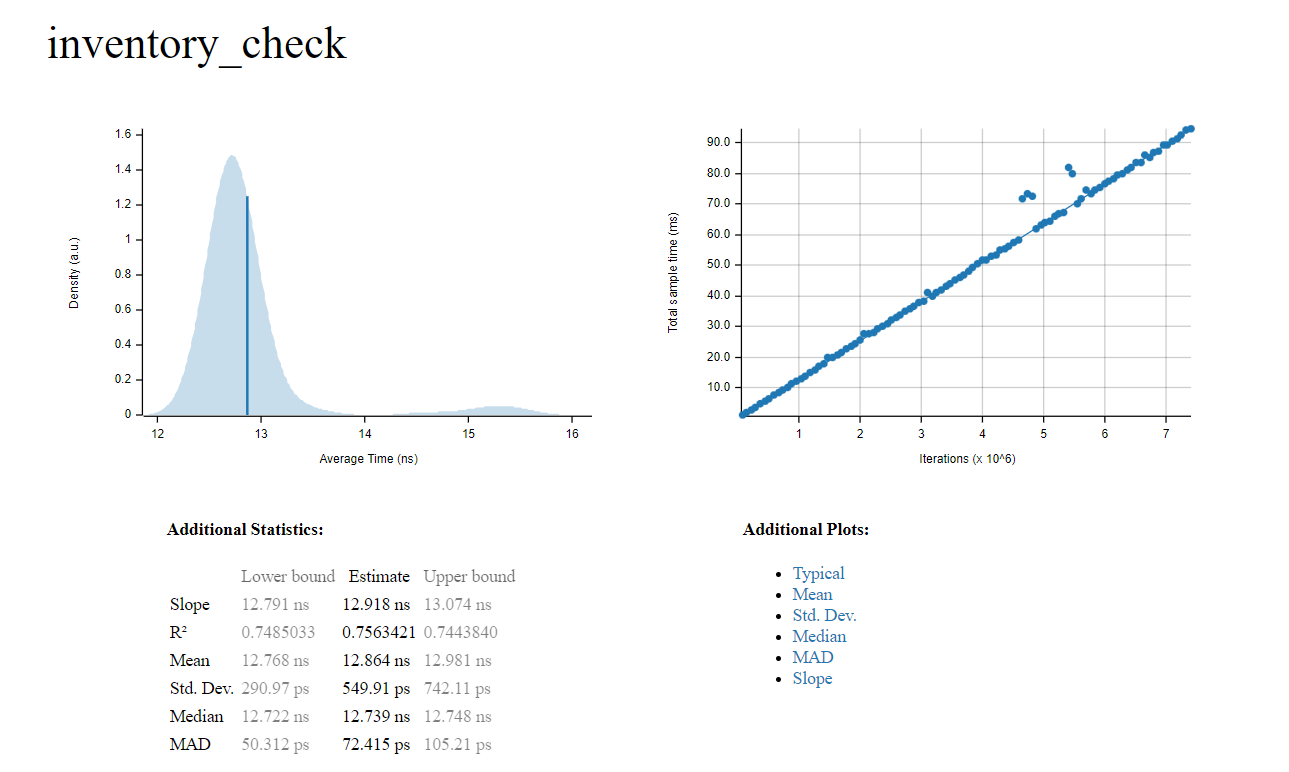


Figure 33 Benchmark 4 – Criterion Report

For the "inventory\_check" function, the R² value of approximately 0.7563421 indicates a moderate positive correlation between the number of iterations and the total execution time, suggesting that as the number of runs increases, there is a tendency for the execution time to increase as well, although the correlation is not very strong. The mean average time for each run is estimated to be around 12.864 nanoseconds, with a median of approximately 12.739 nanoseconds. The mean absolute deviation (MAD) is 72.415 picoseconds, indicating minimal deviation from the average time, while the standard deviation is 549.91 picoseconds, suggesting some dispersion of times around the mean. Overall, these metrics suggest relatively consistent performance with some variability in execution times, highlighting moderate stability in performance for the "inventory\_check" function.

## 5.5 Benchmark 5 – Scheduled Inventory Report

A screen shot of a computer code

Description automatically generated

Figure 34 Benchmark 5 – Function

This “scheduled\_inventory\_report” function is a scheduled task using ScheduledThreadPool, and by using executing\_at\_fixed\_rate(), every 20 seconds, it will generate an inventory report that showcase the inventory status at that moment.

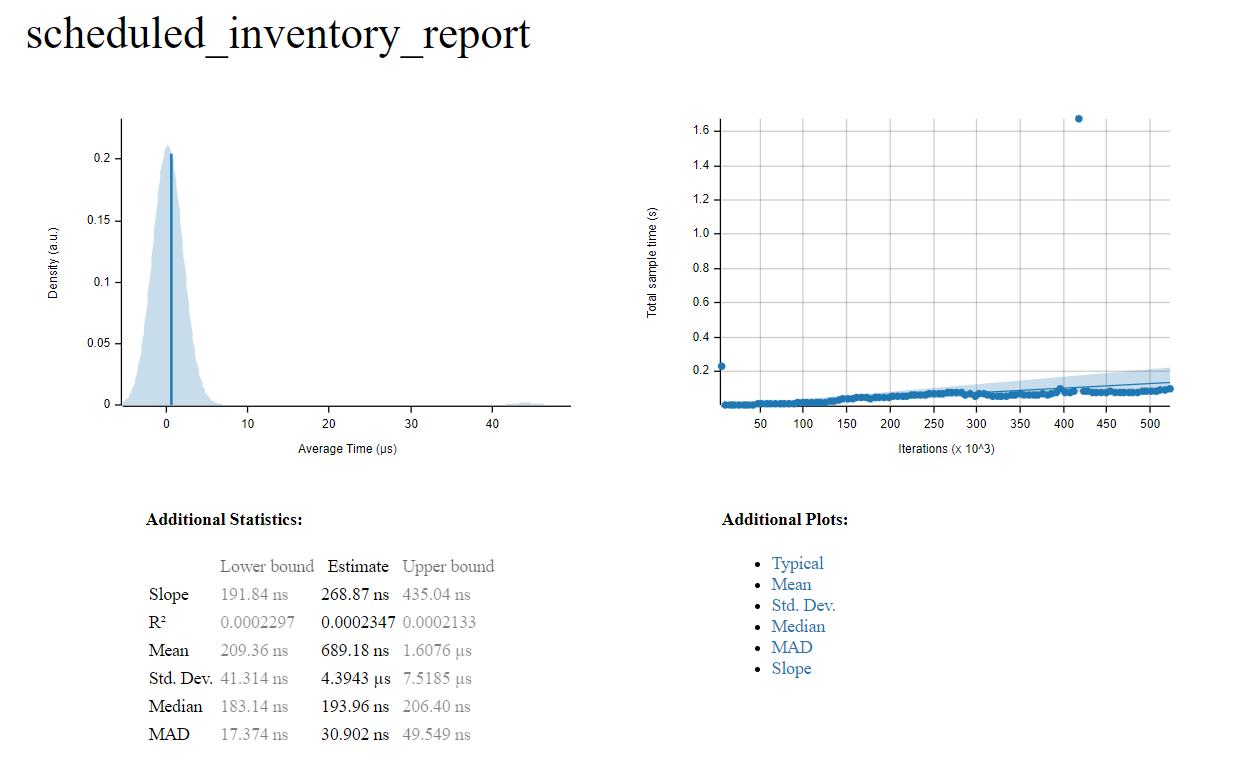


Figure 35 Benchmark 5 – Criterion Report

For the "scheduled\_inventory\_report" function, the R² value of approximately 0.0002347 suggests a very weak correlation between the number of iterations and the total execution time, indicating that changes in the number of runs have minimal impact on the execution time. The mean average time for each run is estimated to be around 689.18 nanoseconds, with a median of approximately 193.96 nanoseconds. The mean absolute deviation (MAD) is 30.902 nanoseconds, indicating relatively low deviation from the average time, while the standard deviation is 4.3943 microseconds, suggesting some dispersion of times around the mean. These metrics imply that the performance of the "scheduled\_inventory\_report" function is characterized by considerable variability in execution times, with a very weak correlation between the number of iterations and total execution time

# 6.0 Conclusion

In conclusion, this individual assignment demonstrates the feasibility of using Rust for building complex real-time systems such as warehouse management. By leveraging Rust's strong concurrency support and external libraries, we were able to develop a robust and efficient simulation. The implementation effectively replicates real-world warehouse operations, including order processing, goods transportation, and inventory management. The use of multithreading, communication channels, and message queues ensures seamless interaction between system components, enhancing overall performance and scalability. Through benchmarking and analysis, we gained valuable insights into function performance, enabling further optimization and refinement. Overall, this project serves as a testament to Rust's capabilities in developing high-performance, concurrent systems for real-time applications.

# 7.0 References

Balasubramanian, A., Baranowski, M., Burtsev, A., Panda, A., Rakamari, Z., & Ryzhyk, L. (2017). System programming in rust. *Operating systems review*, 94-99.

Bugden, W., & Alahmar, A. (2022). Rust: the programming language for safety and performance. *Rust: the programming language for safety and performance*.

Cedeño, W., & Laplante, P. (2007). An overview of Real-Time operating systems. *JALA*, 40-45.

Chen, H. (2014). A real-time tasks scheduling method based on dynamic priority. *Journal of circuits, systems, and computers*.

Crichton, W., Gray, G., & Krishnamurthi, S. (2023). A grounded conceptual model for ownership types in Rust. *Proceedings of the ACM on programming languages*, 1224-1252.

Li, X., Ding, H., Lee, K., Sha, L., & Caccamo, M. (2006). Feedback fault tolerance of real-time embedded systems. *SIGBED review*, 23-28.

Obregon, A. (2024). *Rust: Safe Systems Programming | Medium*. Retrieved from Medium: https://medium.com/@AlexanderObregon/rust-for-systems-programming-what-makes-it-stand-out-3500ef15a25e#:~:text=Memory%20Safety,-A%20defining%20feature&text=Unlike%20other%20system%20programming%20languages,ensuring%20more%20efficient%20memory%20usage.&tex

Pearce, D. J. (2021). A lightweight formalism for reference lifetimes and borrowing in rust. *ACM transactions on programming languages and systems*, 1-73.

Rammig. (2003). *Modeling of Dynamically Modifiable Embedded Real-Time Systems.* 2003 The Ninth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems.

Stoyenko, A. D. (1992). The evolution and state-of-the-art of real-time languages. *Journal of systems and software*, 61-83.

Tripathi, S., Yadav, R., Ranvijay, & Jana, R. (2011). Resource Management in Real Time Distributed System with Security Constraints. *International journal of distributed systems and technologies*, 38-58.

Xuhui, ChenZhuangbin, SunMingshen, ZhouYangfan, & LyuMichael. (2021). Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs. *ACM transactions on software engineering and methodology*, 1-25.

Yu, Z., Song, L., & Zhang, Y. (2019). *Fearless concurrency? Understanding concurrent programming safety in Real-World Rust software*. Retrieved from arXiv.org: https://arxiv.org/abs/1902.01906