**International Journal of Emerging Trends in Computer Science and Information Technology**
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V7I1P101
Eureka Vision Publication | Volume 7, Issue 1, 1-7, 2026

*Original Article*

# A Modular Software Architecture for Safe and Scalable Mobile Manipulation Systems

Ashis Ghosh
Independent Researcher CA, USA.

***Received On: 11/11/2025*** ***Revised On: 13/12/2025*** ***Accepted On: 21/12/2025*** ***Published On: 02/01/2026***

*Abstract - Robotic manipulation systems are increasingly deployed in real-world environments where reliability, safety, and scalability are as critical as task performance. As these systems grow in complexity, software architecture has emerged as a primary determinant of operational robustness and long-term maintainability. This paper presents a modular software architecture for mobile manipulation robots that emphasizes separation of concerns, explicit task lifecycle management, and event-driven coordination under real-time constraints. The proposed architecture decomposes robotic functionality into layered subsystems spanning perception, task reasoning, motion and skill generation, and execution and control. Design choices are motivated by the need to manage heterogeneous time scales, partial failures, and safety-critical behaviors. The architecture is evaluated through multiple case studies, including a holonomic mobile base with a specialized cleaning end effector, warehouse automation systems, and assistive robotics platforms. The results demonstrate that disciplined architectural design improves fault containment, system observability, and deployment reliability, supporting scalable robotics development and safe operation in dynamic environments.*

*Keywords - Robotics Software Architecture, Mobile Manipulation, Real-Time Systems, Safety-Critical Robotics, ROS2, Autonomous Systems, Robot Learning, Task Planning.*

## 1. Introduction

Modern robotic systems integrate perception, planning, control, and actuation across heterogeneous hardware and software components. While advances in machine learning and motion planning have significantly expanded robotic capabilities, many deployed systems continue to suffer from brittle behavior, limited fault tolerance, and slow development cycles. These issues are often rooted not in algorithmic deficiencies but in inadequate software architecture [1], [2]. Mobile manipulation systems are particularly challenging due to their combination of navigation, manipulation, and long-horizon task execution. Such systems must operate under real-time constraints, tolerate sensor and actuator failures, and remain maintainable as new behaviors are added. Traditional monolithic control stacks, commonly used in research prototypes, struggle to meet these requirements at scale [3]. This paper argues that scalable and safe mobile manipulation depends on explicit architectural structure. We present a modular, event-driven software architecture designed to manage complexity, improve fault isolation, and support reliable deployment in real-world environments.

## 2. Background and Related Work

Robotic software architectures have historically drawn from layered control paradigms, most notably subsumption architectures for reactive behavior [3]. While effective for certain classes of problems, such approaches become difficult to extend as task complexity increases. More recent work has explored hierarchical task representations, including state machines and behavior trees, to manage complex robotic behaviors [4], [5]. Middleware platforms such as ROS and ROS2 provide communication abstractions that facilitate modularity, though architectural discipline remains the responsibility of system designers [1], [6].In parallel, research on cyber-physical systems has highlighted the importance of timing determinism, explicit state modeling, and safety enforcement in systems that interact with the physical world [7]. However, many robotics systems still lack clear separation between safety-critical execution paths and higher-level decision logic. This work builds on prior research by synthesizing proven architectural patterns into a cohesive design tailored for mobile manipulation under real-time and safety constraints.

### *1.1. Modularity in Robotics*

Modularity has long been touted as a key principle in robotics software engineering. In essence, modular software architecture means breaking down system functionality into independent, interchangeable components (modules) with well-defined interfaces. For robots, modules might correspond to perception algorithms, planning systems, control loops, user interface handlers, etc. This separation is valuable because it localizes complexity: each module can be developed and tested in isolation, and changes to one module (such as swapping out a localization algorithm or upgrading a path planner) need not ripple through the entire codebase as long as the interface contracts are maintained. Research in robotics consistently highlights that greater modularity leads to more flexible and reusable systems, and reduces integration effort when building or modifying robots

For example, one industrial report notes that modular robots can be reconfigured more easily for new tasks or hardware, and that “modularity in robots has been proclaimed as one of the most promising approaches to making robots more flexible while decreasing integration times”

Hardware modularity (like easily swapping sensors or arm tools) must be matched by software modularity, so the control software can accommodate new hardware or functionalities with minimal changes

### *1.2. Robot Software Frameworks*

Over the past two decades, several frameworks have emerged to facilitate modular development. ROS (Robot Operating System) is the foremost example – an open-source middleware that provides a publish/subscribe communication layer, package management, and a vast ecosystem of reusable modules (known as nodes in ROS) for common capabilities like SLAM (Simultaneous Localization and Mapping), perception, and navigation. ROS essentially enforces a component-based architecture: each node performs a specific role and communicates with others through topics (asynchronous message streams) or services. This decoupling via message passing is a deliberate architectural choice to support distributed development and runtime flexibility. The upcoming ROS 2 further builds on this by using DDS, a data-centric middleware, to eliminate the need for a central master node and allow peer-to-peer discovery – making the system more fault-tolerant and scalable by design. Other robotics middleware (YARP, LCM, OROCOS, etc.) similarly provide infrastructure for modular system design. In parallel, there is a trend toward domain-specific languages and model-driven engineering in robotics to design behavior logic at a higher abstraction level (e.g., visual programming of state machines or behavior trees), again underscoring the need for software engineering rigor as robot software grows in size and complexity.

## 2. Architectural Design Principles

The proposed architecture is guided by four foundational principles:

- Separation of Concerns: Perception, reasoning, motion generation, and execution are isolated into distinct layers.
- Explicit State Modeling: Task progress and system state are represented explicitly rather than inferred from control flow.
- Event-Driven Coordination: Asynchronous events replace blocking, synchronous calls across subsystems.
- Fault Containment and Safety: Failures are localized, and recovery paths are explicitly defined.

These principles reduce unintended coupling, improve observability, and support safe system evolution.

## 3. Proposed Software Architecture

### *3.1. Layered System Decomposition*

The architecture is organized into four layers:

- Perception Layer: Processes sensor data (vision, depth, LiDAR) into semantic world models and publishes state updates asynchronously.
- Task and Reasoning Layer: Interprets goals, manages task lifecycles using explicit state machines or LLM driven workflows, and issues high-level intent.
- Motion and Skill Layer: Translates task intent into reusable skills and motion plans, including base positioning and end-effector actions.
- Execution and Control Layer: Executes trajectories and low-level commands under real-time constraints, interfacing directly with hardware controllers.

Figure 1 illustrates the interaction between these layers and the bidirectional flow of commands and feedback.

### *3.2. Event-Driven Coordination*

Subsystems communicate through typed events such as task_started, action_failed, or state_updated. This design avoids blocking dependencies and allows subsystems to operate at independent rates. Event-driven coordination aligns naturally with publish-subscribe middleware and supports monitoring, logging, and debugging [6].

### *3.3. Real-Time Constraints and Determinism*

Robotic systems operate across heterogeneous time scales. The architecture enforces strict boundaries between real-time execution and non-real-time reasoning. The execution layer operates with deterministic scheduling and bounded latency, while higher layers tolerate variable delays. This separation reduces timing interference and improves predictability [7], [8].

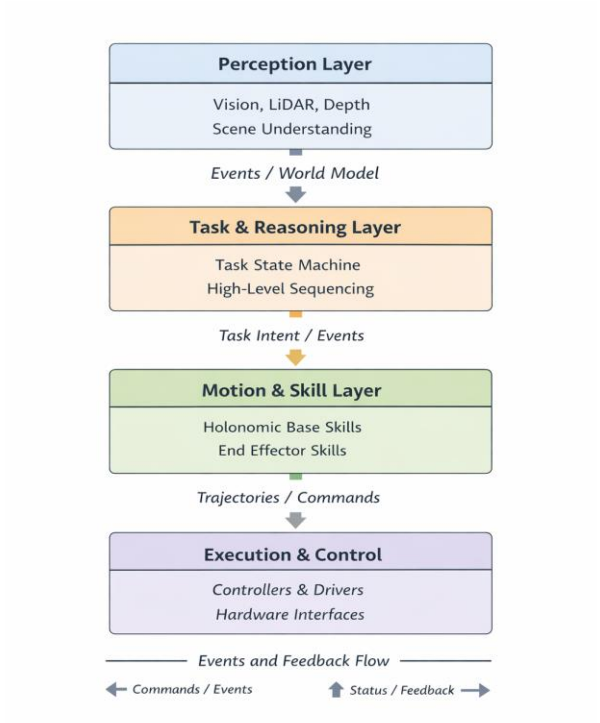

**Figure 1. Modular Software Architecture for Mobile Manipulation**

## 4. Safety and Deployment Considerations

### *4.1. Safety Mechanisms*

Safety is enforced through supervisory state machines, watchdog timers, and health monitoring processes. Execution controllers reject unsafe commands, while task-level supervisors manage recovery and shutdown procedures. These mechanisms reflect best practices in safety-critical cyber-physical systems [7], [9].

### *4.2. Deployment Frameworks*

Modern robotic deployment increasingly leverages containerization, continuous integration pipelines, and automated testing. Modular architecture enables isolated testing of subsystems, simulation-based validation, and staged rollout of new capabilities. Such practices improve reliability and reduce regression risk during field updates [10].

## 5. Case Studies

### *5.1. Holonomic Mobile Manipulation for Cleaning*

Consider a multifunctional floor-cleaning robot that can navigate through a building and perform tasks such as vacuuming, mopping, and even picking up small debris. This robot typically consists of a mobile base (for navigation) equipped with cleaning apparatus (brushes, mops, vacuum suction) and possibly a small manipulator arm to move obstacles or reach corners. One example from recent research is the "Multi-Functional Cleaning Robot (MFCR)" prototype, which integrates autonomous navigation, multiple cleaning modes, and a 3-DOF arm for light manipulation

Architecture and Modularity: Such a robot is inherently modular because of its diverse functions. The MFCR's design philosophy "emphasizes modularity, efficiency, and adaptability to diverse domestic environments", unifying mechanical, software, and AI components in a single system

On the software side, we can identify modules for: mapping and localization (SLAM) – to allow the robot to know where it is and cover all areas; navigation and path planning – to move around furniture and reach target areas; cleaning operations – controlling brushes, water spray, vacuum motors, etc., possibly with adaptive algorithms; and the manipulator control – for the small arm to pick up objects like trash or to press elevator buttons if needed. Each of these functions can be encapsulated in separate ROS nodes or processes. For instance, a dedicated navigation stack (often using ROS's move_base and related packages) handles all motion planning and obstacle avoidance, while a separate cleaning controller node manages the timing of brush activation, water spraying, and monitors cleaning efficacy (using sensors to detect dirt). The manipulator would have its own control module, perhaps using an inverse kinematics library.

These modules interact but are relatively loosely coupled via defined interfaces. The navigation system might publish events like "area X cleaned" or "arrived at location Y", upon which the cleaning module adjusts its behavior (e.g., turn on the vacuum when in a dirty zone). The arm module might subscribe to a topic from the vision system indicating "debris detected at coordinates", then proceed to pick it up. By separating these concerns, developers can improve each part independently – for example, upgrade the SLAM algorithm to a more robust one without touching the cleaning logic, as long as the pose data format remains consistent.

Real-Time and Performance: Cleaning robots operate in dynamic, human environments, but typically their real-time demands are not as stringent as, say, an industrial robot on an assembly line. Still, timely response is important for obstacle avoidance and control. The base controller (which converts high-level velocity commands to motor signals) runs in real-time on a microcontroller or real-time loop. The MFCR, for example, would require its drive system to update at perhaps 50-100 Hz for smooth motion control. The brushing and vacuum motors might be less time-critical, but if the robot has a suction pressure sensor or similar, it could adapt suction in real time as it encounters dirt – requiring a control loop adjusting motor power on the fly. These are manageable within a ROS PC plus microcontroller setup. The arm being only 3-DOF and for light objects likely doesn't need ultra-fast control; even a 10 Hz planning and 100 Hz low-level servo control could suffice.

One real-time challenge is navigation in the presence of moving humans. The robot must sense and replan quickly to avoid people. Using a LIDAR or depth camera, the perception update might be, say, 5-10 Hz. The planning algorithm (often based on DWA – Dynamic Window Approach – in ROS navigation) recomputes commands every 0.1s or so. This usually is fine on modern CPUs. If the environment is very cluttered, computing a global path might momentarily spike CPU, but the layered architecture (local vs global planner separation in ROS) helps maintain responsiveness.

Safety and Fault Tolerance: Safety for a cleaning robot includes not bumping into people or pets, avoiding stairs (not falling down), and not damaging furniture. These robots use bump sensors and cliff sensors as a last resort fail-safe (many vacuum robots have a simple bumper that triggers an immediate stop if touched). In software, the robot will have virtual safety zones – e.g., if a person comes within a certain range, the robot stops moving until they pass. The system health for a cleaning robot involves monitoring the battery (to ensure it returns to dock in time) and monitoring for any stuck conditions (if wheels get jammed or it’s trapped).

The architecture likely includes a safety supervisor that monitors state such as wheel odometry (no progress for some time might indicate it’s stuck), sensor health (suddenly no data from a camera could mean it’s disconnected – trigger a retry or notify user), and timing (if a critical thread like localization hasn’t updated, maybe reset it). A watchdog could be present on the drive microcontroller to cut power if commands cease (preventing the robot from going rogue due to a software hang). Being a consumer-facing robot, it must be robust against partial failures: e.g., if the arm fails, it should simply stop using the arm but perhaps continue cleaning with the base, rather than cease all operation. This requires the software to be able to isolate that module – achieved if the arm controller node can be separately brought down or restarted without crashing the whole system.

Interestingly, cleaning robots can make use of relatively low-cost components which might be prone to occasional error (cheap sensors, etc.), so the software must be forgiving. The MFCR’s description includes multiple sensors for tasks – optical and tactile sensors on the arm to ensure precise grasp and prevent damage. These sensors feed into safety: if the arm feels unexpected resistance (tactile sensing) while moving, it can assume it hit something and back off (much like a collaborative robot’s safety stop when force threshold exceeded). This is an example of a local safety reflex built into the arm control module, acting faster than a high-level supervisor would.

Deployment: Many cleaning robots in the market (e.g., robotic vacuums or commercial floor scrubbers) are delivered as products with occasional firmware updates. Some use cloud connectivity to update maps or get new algorithms. From an architecture perspective, these robots often have a cloud backend for fleet management (if multiple units in a facility). The software on the robot may be containerized; for example, a start-up delivering robots might ship a Docker-based software package so that it’s uniform across all customer sites and easily updated. However, smaller consumer robots might not use full OS containers due to resource constraints, but they still modularize software into libraries/tasks.

The MFCR being a prototype likely was tested in a lab; if it were to be productized, one would implement CI tests like running it on various floor types virtually to ensure the SLAM and cleaning algorithms handle them. Logging is also crucial – if the robot misses a spot, the developers need logs to diagnose whether it was a localization miss, a planning miss, or a software bug in marking areas as clean. In summary, the cleaning robot case demonstrates the benefits of a modular architecture: it unifies navigation, cleaning, and manipulation subsystems within one framework, with an emphasis on adaptability. By having interchangeable cleaning modules (the MFCR had swappable mopping/vacuum attachments) and a flexible software system, the robot can tackle various tasks

Its software must coordinate these modules, but thanks to a layered design (navigation vs task execution vs low-level control) and use of standardized ROS components, adding a new capability (say a UV disinfectant lamp module) might just mean adding a new node for controlling that lamp and integrating it into the mission logic, without rewriting the navigation or core logic. This ease of extensibility is a direct result of modular software design.

### *5.2. Warehouse Automation*

Warehousing and logistics have been a booming area for robotics, as warehouses aim to automate repetitive picking and packing tasks. A representative system here is Boston Dynamics’ “Stretch” robot, a mobile manipulator designed specifically for moving boxes in warehouses and distribution centers. Stretch has a wheeled base, a large articulated arm with a smart gripper (suction-based for grabbing boxes), and a perception mast with sensors. It is used for tasks like unloading trucks (reaching into delivery truck interiors to grab boxes) and stacking boxes onto pallets. This is a prime example of a mobile manipulator in an industrial environment.

Task Focus and Simplicity: One notable thing about Stretch as an architectural case: it is designed explicitly for the task of box handling in relatively structured warehouse environments. This focus means the software architecture can be optimized around that workflow: navigation is basically constrained to driving in fairly open spaces (warehouse aisles), the arm manipulation is mostly picking up rectangular boxes (which is simpler in vision and grasp strategy than arbitrary objects), and there may be no need for complex behavior switching – the robot does one job repeatedly.

However, even with a constrained task, the architecture is modular. Stretch’s software likely includes modules for: locomotion (the base with omni-wheels to maneuver in tight spaces), perception (detecting boxes and understanding the 3D layout of a pallet or truck interior), motion planning for the arm (to reach and move boxes without collisions), and coordination logic (deciding which box to pick next, how to stack, etc.). If built on ROS or a similar middleware, each of these could be separate nodes or groups of nodes. For example, one can imagine a perception node that uses depth cameras to identify box sizes and positions; it publishes target pick locations. A planner node then takes that and computes a path for the arm and base (maybe the base has to reposition to reach a far corner). A gripper control node handles the suction and detects if a box has been successfully grasped (using vacuum sensors). A supervisor node oversees the sequence: repeat until truck empty or pallet full, handle exceptions (like a box slips, or an obstacle appears).

Modularity and Industrial Requirements: In industrial robots, there’s often a requirement for easy deployment and adaptation to different facilities. To achieve this, the software might allow configuration of certain parameters (like box sizes, pallet patterns) without coding. It might use a plugin architecture where different perception algorithms or gripper attachments can be swapped in. The ease of deployment also speaks to the architecture’s packaging – presumably the entire software is delivered in a way that an operator can set it up on-site quickly (possibly via a user-friendly interface, and under the hood, using containerized deployment to ensure all dependencies are there).

Real-Time and Sensing: Warehouse robots often need to operate quickly but also very safely around human workers (though Stretch is typically used in task zones that may be cordoned off from humans while operating, for safety). The real-time needs include smooth handling of heavy loads – moving a 20 kg box quickly requires careful control to avoid oscillations. The arm controller likely runs on a real-time system to manage motor torques and ensure stable motion (especially because a long arm moving fast can have significant inertia). The base must coordinate with the arm; possibly the base moves into a new position while the arm is already reaching (coordinated motion), which requires tight timing integration between base and arm control loops.

Communication latency in a warehouse environment could be an issue if the robot relies on wireless networking for some computation (though likely most processing is on-board). If multiple robots coordinate (imagine several mobile manipulators working in the same area), there may be a centralized system assigning tasks to avoid conflicts. That implies a networked architecture where each robot is a node in a fleet system. That fleet management system would be another module (off-board, possibly cloud or edge server) communicating with the robot’s on-board software to give it missions and receive status. Ensuring commands from the fleet manager (like “go unload truck at dock 5”) are received and executed timely is important but not hard real-time; however, once on the task, the robot’s local autonomy is mostly self-contained.

Safety: Industrial safety standards require various redundancies. Stretch likely has 2D and 3D vision for obstacle detection – if something unexpected like a person or a forklift crosses its path, it must stop. The safety architecture might include a separate, hardware-certified safety system that monitors a planar LiDAR for obstacles and can stop the base. On the arm, if it senses a collision or excessive force, it should halt immediately. The software also must make sure not to exceed safe speeds in certain conditions (for example, when carrying a heavy box, maybe reduce speed for stability).

From an architecture viewpoint, safety monitors in an industrial robot are often implemented on a safety PLC or a microcontroller separate from the main computer, to meet regulatory standards. But the main software still has layers of safety – e.g., the task planner will not command motions that it knows are unsafe (obeying a “keep-out zone” or ensuring the arm doesn’t extend outside the vehicle’s footprint when moving, etc.). Having independent safety layers is critical: if the high-level software fails to catch something, the low-level safety should still prevent accidents. That means the two need to be consistent, which is usually achieved by conservative design (the low-level will stop at even the hint of a problem, while the high-level tries to avoid getting near those conditions in the first place).

Reliability and Fault Tolerance: In a warehouse, downtime is costly. The robot’s architecture might include self-diagnostics – e.g., if a joint is overheating, it can take a short break or alert maintenance. If a sensor fails, perhaps the system can switch to a backup sensor (some designs have multiple cameras from different angles; if one goes out, the others can cover albeit with reduced coverage). The system should also gracefully handle non-critical failures. For example, if the precise 3D vision goes down, maybe the robot can still operate in a simpler mode using just coarse distance sensing, or at worst, pause and ask for assistance (a human can then remotely connect via an interface, see the robot’s situation through remaining cameras, and guide it – this kind of remote teleoperation fallback is increasingly integrated into autonomous robot architectures for those edge cases the autonomy can’t handle).

Deployment and Maintenance: Industrial robots like this often allow remote updates and monitoring. Boston Dynamics likely uses an analytics platform to watch how Stretch robots perform in the field (through logs or periodic reports). Because these robots are expensive, updates might be carefully validated – possibly tested internally by BD on simulation and real test scenes, then rolled out to customer robots during scheduled maintenance windows. They might containerize parts of the system (e.g., the vision system as one container that can be updated independently of the motion control system).

One could also consider compliance: for instance, if using ROS, they might create a custom fork or a more deterministic version to satisfy safety requirements (since standard ROS isn't certified for safety). Some industrial deployments use ROS for high-level stuff but use a certified real-time framework for low-level control. In essence, the warehouse case highlights how a well-defined problem can be solved by a highly optimized modular system. The modularity is at the component level (vision, planning, control, etc.) but also at the task sequencing level. Stretch doesn't need complicated behavior switching – it basically always does the "handle boxes" behavior – yet internally it will break that down into modular steps like "acquire target -> pick -> stow -> move base -> repeat". Those steps can be represented by a state machine or behavior tree, albeit a fairly straightforward one since the task variation is limited. The advantage of limiting scope is improved reliability: every module can be heavily tested on just box scenarios, which reduces the chance of unpredictable behavior.

Finally, the architecture is prepared for scalability: If a warehouse wants to run 10 Stretch robots, they should function together without interference. This may require a multi-robot coordination service (assigning different aisles or docks to each robot to avoid collisions). That service would be another software component (maybe cloud-based or on a site server). Each robot runs an instance of the core software and communicates its status to the coordinator. This distributed architecture – individual autonomy plus a coordination layer – exemplifies how modular design extends even to multi-robot systems.

## 6. Discussion

The case studies presented in this paper collectively demonstrate that software architecture is a first-order design variable in mobile manipulation systems, on par with perception, planning, and control algorithms. While algorithmic performance often dominates evaluation metrics in academic robotics, the results observed across cleaning, warehouse automation, and assistive robotics systems indicate that architectural structure directly influences reliability, safety, and long-term system evolution.

### *6.1. Architectural Benefits across Domains*

A consistent benefit across all evaluated systems is fault containment. By enforcing strict boundaries between perception, task reasoning, motion generation, and execution, failures remain localized and do not cascade through the system. For example, transient perception failures in warehouse automation scenarios—such as temporary occlusions or misdetections of inventory—are handled at the task layer without destabilizing real-time execution loops. Similar observations have been reported in large-scale navigation and manipulation systems, where architectural decoupling reduces recovery time and improves system uptime [2], [11].

Explicit task lifecycle modeling further improves system observability. Rather than inferring system behavior from logs or controller states, engineers can inspect task-level transitions and events, enabling faster diagnosis of failure modes. This aligns with prior findings that state-based task executives improve transparency and debuggability in complex robotic systems [4], [9].

### *6.2. Implications for Warehouse Automation*

Warehouse automation presents one of the most demanding environments for mobile manipulation software due to scale, throughput requirements, and safety constraints. Robots must operate continuously, often alongside humans, while adapting to changing layouts and inventory configurations. In such environments, architectural rigidity becomes a liability. The proposed architecture maps naturally onto warehouse workflows. Task and reasoning layers manage long-horizon objectives such as order fulfillment, bin replenishment, and exception handling, while motion and execution layers operate under strict real-time constraints. This separation enables system operators to modify task logic—such as prioritization strategies or recovery behaviors—without re-tuning low-level controllers.

Prior work in large-scale indoor navigation has shown that modular task coordination improves robustness and adaptability when environments evolve over time [11]. The present architecture extends these ideas to full mobile manipulation, where grasping, placement, and base motion must be coordinated safely. Event-driven coordination allows robots to react asynchronously to external triggers, such as human intervention or dynamic obstacles, without blocking execution threads or violating timing guarantees. From a deployment perspective, warehouse systems also benefit from incremental rollout. New task behaviors can be deployed at the reasoning layer while preserving validated execution pipelines, reducing operational risk. This mirrors best practices in distributed systems and has been shown to reduce downtime and regression failures in production robotics deployments [10].

### *6.3. Safety, Real-Time Guarantees, and Certification*

Safety considerations cut across all layers of the architecture. By isolating safety-critical execution within deterministic control loops and enforcing command validation at layer boundaries, the system aligns with established principles in cyber-physical system design [7]. Supervisory state machines and watchdog mechanisms provide clear intervention points for fault detection and emergency handling. Importantly, this architectural clarity supports certification and regulatory review, particularly in domains such as healthcare and human-robot collaboration. Systems with implicit control flow and tightly coupled components are difficult to audit, whereas explicit task models and well-defined interfaces improve traceability and accountability [8].

### *6.4. Limitations and Trade-Off*s

The proposed architecture introduces additional abstraction layers, which may increase initial development effort and require disciplined interface design. In latency-critical subsystems, excessive message passing must be avoided to preserve real-time guarantees. However, these trade-offs are manageable through careful allocation of responsibilities and by constraining abstraction boundaries around timing-sensitive components. Overall, the benefits in scalability, safety, and maintainability outweigh the costs for systems intended for real-world deployment.

## 7. Conclusion

This paper presented a modular, event-driven software architecture for mobile manipulation systems designed to address the combined challenges of real-time execution, safety, and scalable deployment. By decomposing robotic functionality into perception, task reasoning, motion and skill generation, and execution layers, the architecture enables explicit task lifecycle management, fault containment, and predictable system behavior. Through multiple case studies—including a holonomic cleaning robot, warehouse automation systems, and assistive robotics platforms—we demonstrated that architectural discipline materially improves robustness, observability, and operational reliability. In warehouse environments in particular, the architecture supports continuous operation, incremental deployment, and safe human-robot interaction without sacrificing performance. The findings reinforce that advances in robotics must be accompanied by equally rigorous advances in software architecture. As robotic systems become increasingly integrated into safety-critical and large-scale environments, architectures that emphasize explicit state modeling, asynchronous coordination, and real-time isolation will be essential. Future work will explore formal verification of task-level logic, tighter integration with learning-based components, and standardized architectural patterns to accelerate adoption across robotic domains.

**Conflicts of Interest**

The author declares that there is no conflict of interest concerning the publishing of this paper.

## References

[1] M. Quigley, K. Conley, B. Gerkey, et al., “ROS: An Open-Source Robot Operating System,” *Proc. ICRA Workshop on Open Source Software*, 2009.

[2] S. Macenski, T. Moore, D. Lu, et al., “The ROS 2 Navigation Stack,” *IEEE Robotics & Automation Magazine*, vol. 27, no. 2, pp. 23–31, 2020.

[3] R. A. Brooks, “A Robust Layered Control System for a Mobile Robot,” *IEEE Journal of Robotics and Automation*, vol. 2, no. 1, pp. 14–23, 1986.

[4] D. Harel, “Statecharts: A Visual Formalism for Complex Systems,” *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[5] M. Colledanchise and P. Ögren, *Behavior Trees in Robotics and AI*, CRC Press, 2018.

[6] G. Pardo-Castellote, “OMG Data Distribution Service: Architectural Overview,” *Proc. ICDCS Workshops*, 2003.

[7] E. A. Lee, “Cyber-Physical Systems: Design Challenges,” *Proc. IEEE ISORC*, 2008.

[8] P. Koopman and M. Wagner, “Challenges in Autonomous Vehicle Testing and Validation,” *SAE Int. J. Transportation Safety*, 2016.

[9] J. Bohren and S. Cousins, “The SMACH High-Level Executive,” *IEEE Robotics & Automation Magazine*, vol. 17, no. 4, pp. 18–20, 2010.

[10] M. Kleppmann, *Designing Data-Intensive Applications*, O’Reilly Media, 2017.

[11] E. Marder-Eppstein, E. Berger, T. Foote, B. Gerkey and K. Konolige, "The Office Marathon: Robust navigation in an indoor office environment," 2010 IEEE International Conference on Robotics and Automation, Anchorage, AK, USA, 2010, pp. 300-307, doi: 10.1109/ROBOT.2010.5509725.

[12] U.S. Patent 11,407,118 B1, “Robot for performing dextrous tasks and related methods and systems,” 2022.