AutoDuck - Dual-Mode Autonomous Navigation System

Project by Sahil Virani, Supratik Patel, Esmir Kićo

04/09/2025

This report presents a comprehensive study of the Duckiebot DB21J platform, covering assembly, installation, calibration, preconfigured software, and implementation experiments with both the default Duckietown system [1] and Visual Language Models [5]. Challenges with cross-platform deployment, system latency, and model performance tradeoffs are discussed.

1 Introduction

The Duckiebot DB21J is a compact, modular autonomous robotic platform designed for research and education in the Duckietown environment [1]. This report aims to provide a comprehensive overview of its assembly, installation, calibration, system design, preconfigured software, and experimental evaluation. We employed the 4GB version of the NVIDIA Jetson Nano, providing sufficient computational power for moderate-level inference while maintaining energy efficiency.

Our study focuses on both the default Duckietown stack and additional Vision-Language Model experiments (Qwen 2.5) [5], analyzing performance tradeoffs, system latency, and reliability. The report is structured to provide insights into hardware integration, software configuration, perception, control, and navigation, highlighting practical challenges and optimization strategies encountered during deployment.

2 System Design and Architecture

The Duckiebot DB21J is a compact robotic platform designed for vision-based autonomous driving in the Duckietown environment [1]. It integrates sensing, computation, and actuation components in a modular design, enabling experimentation with perception, control, and navigation algorithms. The following subsections provide an overview of its main hardware components, assembly, and calibration procedures.

2.1 Components

The DB21J Duckiebot used in this project consists of the following major components:

- **NVIDIA Jetson Nano (4GB)**: onboard processing unit executing perception and control algorithms [4].
- **Hardware Utility Board (HUT)**: manages power distribution and communication with sensors and actuators.
- **Camera**: primary perception sensor for lane detection, obstacle recognition, and AprilTag identification [2].
- **DC Motors with Encoders**: provide actuation and wheel odometry for speed and position estimation.
- **Time-of-Flight (ToF) Sensor**: measures distance to nearby objects for obstacle detection.
- **Battery Pack with Diagnostics**: powers the robot and provides system health monitoring.
- **Display and RGB LEDs**: feedback on system state, connectivity, and operation modes.
- **External Wi-Fi Adapter**: enables wireless communication for remote control, monitoring, and data transfer.

2.2 Assembly

The assembly of the DB21J Duckiebot follows official Duckietown guidelines [1]. Key steps include:

- The NVIDIA Jetson Nano connects to the HUT, interfacing with motors and ToF sensor.
- The camera is mounted on the front bumper for an unobstructed forward view.
- The battery connects to the HUT to supply power.
- OLED display and RGB LEDs provide real-time diagnostics.
- The Wi-Fi adapter dongle is connected to the Jetson Nano for wireless operation.

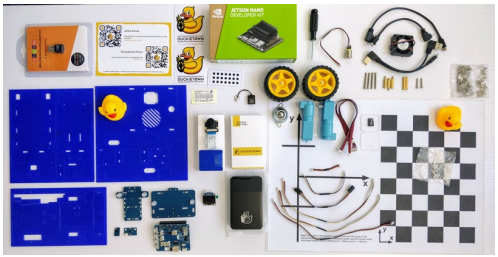


Figure 2.1: *DB21J Duckiebot Assembly (adapted from Duckietown documentation [1]).*

3 Preconfigured System

The Duckiebot DB21J is delivered with a preconfigured software environment designed to accelerate development and ensure reproducibility [1]. The stack integrates Ubuntu Linux, ROS [4], Docker containers, and the `dt-core` repository. This environment provides a consistent baseline, allowing focus on perception, planning, and control.

3.1 Operating Environment

Ubuntu with ROS serves as middleware [4]. ROS allows modular nodes, topics, services, and parameters. Subsystems—lane detection, obstacle detection, motor control—communicate in real time for flexibility, scalability, and debugging.

3.2 Duckietown Framework

The Duckietown stack uses Docker containers to isolate dependencies [1]. Relevant containers:

- **ros**: hosts perception and control nodes.
- **duckiebot-interface**: manages motors, LEDs, camera, ToF.
- **device-health** and **device-online**: diagnostics and connectivity monitoring.
- **dashboard**: browser-based system monitoring.

3.3 Core Software Packages

- **lane_filter** and **lane_control**: estimate lane position and regulate steering/velocity [3].
- **stop_line_filter**: detects red stop lines for halting.
- **apriltag**: detects AprilTags and estimates pose [2].
- **fsm**: finite state machine for mode transitions.
- **vehicle_detection**: identifies obstacles such as duckies.

3.4 ROS Architecture

- **Nodes:** independent processes for drivers, filters, controllers.
- **Topics:** continuous data streams (images, odometry, control commands).
- **Services:** request-response calls (resetting states, fetching calibration).
- **Parameters:** configuration variables (lane gains, camera calibration).

3.5 Workflow and Development Tools

Git repositories and SSH access to the Duckiebot were used for development. Code was deployed in Docker containers for consistency. The `dts` (Duckietown Shell) simplified building containers and launching nodes. Custom scripts automated synchronization and container restarts.

3.6 System Workflow

Camera images are processed by perception modules (anti_instagram), integrated by the FSM with ToF inputs. Decisions pass to the lane controller and then to DC motors. Wi-Fi/ROSBridge enables monitoring.

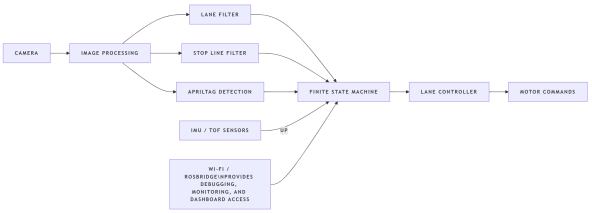


Figure 3.1: *System workflow of the Duckiebot. Perception modules feed into the FSM, integrating sensors before passing decisions to the controller and motors.*

3.7 Limitations and Adaptations

- Lane following tuned for curved tiles.
- Stop line detection enhanced with distance-based stopping.
- Duckie detection improved via color filtering and bounding boxes.
- AprilTag detection calibrated for accuracy.
- Vision–Language Model integration highlighted Jetson Nano hardware limits.

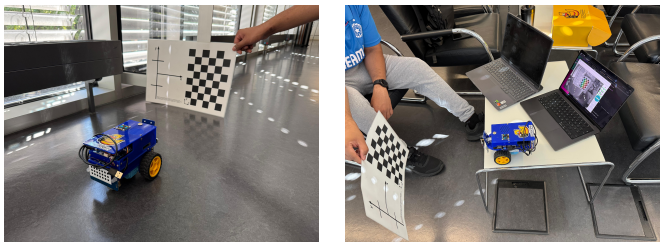
3.8 Summary

The preconfigured Duckietown system provided a reliable baseline with modular Docker containers and ROS nodes. It enabled safe extension of perception and control while supporting prototyping and collaboration.

4 Calibration of Camera and Motors

4.1 Camera Calibration

The DB21J camera requires intrinsic and extrinsic calibration for accurate perception. Intrinsic corrects lens distortions; extrinsic aligns camera frame with robot coordinates. Checkerboard patterns at multiple distances/angles produce camera matrices and distortion coefficients. Proper calibration improves lane following and AprilTag detection.



(a) *Calibration pic1* **(b)** *Calibration pic2*

Figure 4.1: *Camera calibration results.*

4.2 Motor Calibration

Each motor's gain is adjusted for expected velocity and trim values corrected for asymmetry. Iterative calibration ensures straight-line driving. Encoder feedback fine-tunes performance for long-distance accuracy.

5 Implementation

The implementation phase of the Duckiebot DB21J project focused on deploying autonomous navigation behaviors using both the preconfigured Duckietown software stack and experimental Vision-Language Model (Qwen 2.5) integration.

5.1 Lane Following

Lane following is the core behavior for autonomous navigation. The `lane_filter` and `lane_control` modules estimate the robot's lateral position and heading within the lane and generate steering and velocity commands. Performance tuning included adjusting controller gains and filtering parameters to minimize oscillations and improve trajectory stability, particularly on curved tiles. Accurate camera calibration and iterative motor trim adjustments were essential to maintain smooth and precise lane following.

5.2 Stop Line Detection

Intersection handling is managed by the `stop_line_filter` module, which detects red stop lines. During implementation, default parameters occasionally caused premature stopping. To improve robustness, distance-based stopping logic was integrated, allowing the Duckiebot to decelerate gradually and halt at the correct location. This enhancement ensures safe intersection navigation and supports coordination in multi-robot scenarios.

5.3 AprilTag Localization

AprilTags provide semantic localization for precise navigation and decision-making. The `apriltag` module detects tags and estimates their position relative to the robot [2]. This capability is crucial for identifying intersections and key map points. Reliable detection requires well-calibrated camera intrinsics and extrinsics, ensuring accuracy across varying lighting conditions and perspectives.

5.4 Finite State Machine Integration

A finite state machine (FSM) orchestrates the overall navigation, integrating inputs from lane detection, stop line detection and AprilTag localization. The FSM manages state transitions such as idle, lane following, stopping, and recovery. Sensor fusion, including ToF measurements, enhances safety and ensures robust decision-making. The modular design allows perception, planning, and control to operate concurrently while maintaining a clear priority hierarchy.

5.5 Qwen 2.5 Vision Language Model

To explore higher-level perception and reasoning, the Qwen 2.5 model was employed for visual input processing and command generation [5]. Two model sizes were evaluated: 7B and 3B. The 7B model provided precise predictions for trajectory planning and obstacle interpretation but introduced high latency (6–7 seconds per frame), limiting real-time feasibility. The 3B model responded faster (approximately 2 seconds per frame) but with slightly reduced accuracy. The use of different visual language models emphasizes the trade-off between model complexity and real-time responsiveness.

5.6 Vision Language Model Integration

Beyond classical perception, we investigated lightweight *vision–language models* (VLMs) on resource-constrained hardware. Concretely, we evaluated **Qwen2.5-VL**, **Gemma 3 (vision)**, **LLaVA**, and **SMOL-VLM**. Our most stable end-to-end pipeline used a quantized GGUF build of **Qwen2.5-VL-7B-Instruct** executed via `llama.cpp` for faster on-device inference. We customized model settings to our available GPU/VRAM budget (context window, batch size, and quantization) while keeping the runtime simple to maintain reliability in the DB21J workflow.

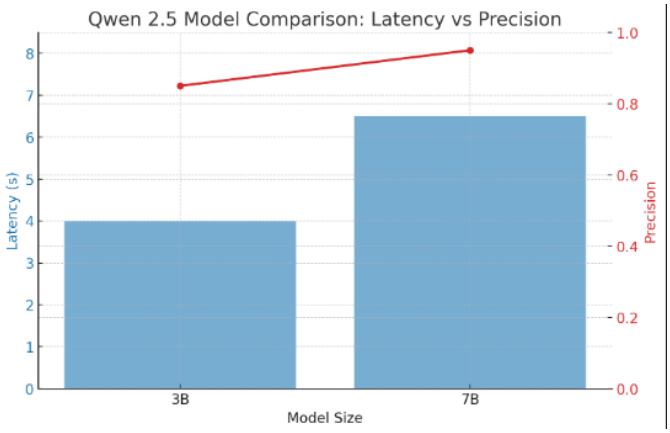


Figure 5.1: *Comparison of Qwen 2.5 model performance: 7B vs 3B. The 7B model shows higher accuracy with increased latency, whereas the 3B model is faster with slightly reduced precision.*

5.7 Challenges

The platform constraints limited real-time VLM use:

- **Memory/VRAM limits:** larger VLMs exceeded available memory without aggressive quantization.
- **Latency:** even with GGUF quantization, image-encoder compute kept loop times above control-cycle targets.
- **Hardware variability:** acceleration support differed across devices; some builds expected CUDA features that were unavailable on our setup.
- **Container reliability:** among tested images, only the GGUF-based Docker image run executed the full vision and language path reliably.

5.8 Insights and Future Work

Although we did not reach real-time VLM operation on-board, two directions look promising:

- **Smaller/quantized VLMs:** prefer GGUF quantizations with `llama.cpp` backends and tune context/batch to the device.
- **Split/hybrid execution:** offload vision-language reasoning to an edge server while keeping low-level perception and control on the DB21J.

5.9 System Performance and Optimization

Extensive testing revealed that while the default Duckietown stack provides a reliable foundation, achieving optimal performance requires careful calibration, parameter tuning, and, when using large models, optimization strategies such as quantization or pruning. Continuous monitoring through ROS topics and the `dashboard` enabled rapid detection of anomalies and iterative improvements.

5.10 Summary

The implementation phase demonstrated the successful integration of perception, decision-making, and control on the DB21J platform. Core behaviors—including lane following, stop line detection, and AprilTag localization—were effectively combined using ROS nodes and the FSM. Vision Language Models offer potential for higher-level reasoning, but hardware limitations necessitate a careful balance between precision and responsiveness.

6 Conclusion

The Duckiebot DB21J proves to be a robust and versatile platform for autonomous navigation research, offering both a reliable baseline for experimentation and significant opportunities for innovation [1]. Through our investigations, it is evident that while the default software provides consistent performance, the system's full potential is unlocked only through careful calibration, sensor optimization, and software tuning. These enhancements are critical for achieving real-time, high-precision operation in dynamic Duckietown environments.

Our work demonstrates the importance of balancing computational complexity and performance. High-capacity models, such as the 7B variant of Qwen 2.5, deliver superior trajectory planning and obstacle interpretation but at the cost of increased latency, limiting real-time responsiveness [5]. Conversely, lighter models like the 3B variant provide faster inference while maintaining acceptable accuracy, highlighting the trade-offs inherent in deploying advanced AI models on resource-constrained robotic platforms.

Furthermore, the integration of vision–language models into the Duckiebot pipeline opens new avenues for multimodal understanding and command generation. Our experiments indicate that with proper quantization and end-to-end pipeline management, these models can operate reliably on embedded hardware, expanding the scope of real-time autonomous decision-making.

Looking forward, continued optimization of perception, planning, and control modules will be crucial to enhance system robustness under varied environmental conditions. Future work may involve incorporating adaptive learning algorithms, more sophisticated sensor fusion, and improved energy management to extend operational efficiency. Additionally, the Duckiebot platform offers a scalable testbed for benchmarking emerging autonomous navigation strategies.

In conclusion, by combining careful hardware calibration, optimized model selection, and innovative multimodal perception, it is possible to achieve a highly responsive and reliable autonomous navigation system. The insights gained from this work provide a foundation for advancing intelligent robotic systems capable of operating effectively in real-world, dynamic settings.

References

[1] *Duckietown Documentation*. `https://docs.duckietown.org`. Accessed: 2025-09-04.

[2] Edwin Olson. "AprilTag: A robust and flexible visual fiducial system". In: *IEEE International Conference on Robotics and Automation (ICRA)* (2011), pp. 3400–3407.

[3] Oliver Peters. *Robust Control for Autonomous Mobile Robots*. Springer, 2018.

[4] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *ICRA Workshop on Open Source Software* (2009).

[5] *Qwen 2.5-VL*. `https://github.com/QwenLM/Qwen2.5-VL`. Accessed: 2025-09-04.