Technology Feasibility

10/16/2025

REACH

Reinforcement-learning
for Enhanced Arm Control
in Humans

Dr. Lerner & Dr. Cunha
Mentor: Bailey Hall

Taylor Davis, Victor Rodriguez, Clayton Ramsey, Lucas Larson

Table of Contents

Introduction

Neurological conditions such as stroke, traumatic brain injury, and spinal cord injury affect millions of people worldwide each year, with many survivors experiencing partial or complete loss of upper-limb motor function. These impairments severely limit the ability to perform essential daily activities such as eating, dressing, and personal care, representing both an enormous economic burden and an immeasurable human cost in terms of lost independence and quality of life.

Dr. Zach Lerner, Associate Professor of Mechanical Engineering at NAU, and Prof. Carlo R. da Cunha, Assistant Professor of Electrical Engineering at NAU, have identified the critical need for adaptive robotic assistance systems in rehabilitation. Their research focuses on developing intelligent control systems that can learn and adapt to individual patient needs, making them ideal sponsors for this capstone project. Their expertise in biomechanics and control systems provides essential guidance for developing safe, effective robotic assistance technologies.

While effective rehabilitation is critical for recovery, access to consistent, high-quality therapy remains limited by therapist availability, geographic barriers, and the labor-intensive nature of traditional one-on-one sessions. Robotic-assisted rehabilitation has emerged as a promising solution to scale therapy delivery, yet current systems face significant challenges: they are expensive, difficult to customize, and require extensive manual programming for each patient or task. There is a clear need for adaptive, intelligent robotic systems that can learn to assist patients efficiently and safely while reducing the burden on healthcare providers.

REACH (Reinforcement-learning for Enhanced Arm Control in Humans) is a capstone project developing a simulation-based reinforcement learning framework for training control policies for wearable robotic arms. The system aims to learn safe, natural movements that assist patients in recovering motor function through repeated, adaptive practice. By leveraging simulation-to-real (sim-to-real) transfer techniques, REACH trains control policies in a physics-based virtual environment before deploying them to real hardware. This approach significantly reduces development time, risk, and cost by enabling rapid iteration and testing of movement strategies without physical hardware constraints or safety concerns.

The REACH framework integrates two complementary sensory inputs: proprioceptive data and visual perception. This multimodal approach enables the system to understand both its own state and the surrounding environment, supporting intelligent task execution and adaptive behavior. The result will be a modular, extensible software framework capable of generalizing across different robotic platforms, tasks, and potentially diverse patient needs.

By developing intelligent control policies that learn and adapt, REACH has the potential to revolutionize assistive robotic systems. Rather than requiring extensive manual programming for each individual or task, the framework enables robotic limbs to learn optimal assistance strategies through reinforcement learning. This adaptability allows users to relearn lost motor abilities while the system continuously refines its behavior to match individual movement patterns and preferences, ultimately restoring independence and improving quality of life.

Now that the system has been introduced, the remainder of this document will describe the feasibility in further detail. From here, this document first discusses the high-level technical challenges that REACH will encounter. Next, each one of the technical challenges will be evaluated, using specific criteria to help determine what technology will work best for the given challenge. After doing a full analysis of the technologies, and choices are made for what technologies will be used, a high-level method of integration will be discussed. After this, a feasibility conclusion will be made.

Technological Challenges

The REACH project faces several core challenges that will be determinant of both its feasibility and long-term success.

1. Simulation Environment
   We need a physics simulator that models contact, friction, and joint actuation accurately while staying stable for long training runs. It must integrate cleanly with Gymnasium/SB3 and support fast scene authoring for our arm tasks.
2. Visual Perception
   The system must extract reliable, low-latency visual features from camera frames for control. Our pipeline should pair capture/calibration with lightweight feature extraction that feeds the policy.
3. Proprioception
   We require a high-rate joint state pipeline (positions/velocities/torques) with consistent normalization and low jitter. The path from simulator to policy should favor simplicity and zero-copy handoff into PyTorch.
4. Reinforcement Learning Framework
   The RL stack must provide proven algorithms (PPO/SAC), stable training, and easy customization. SB3 on PyTorch should plug into our sim and accept both vision and proprioceptive inputs with minimal boilerplate.
5. Programming Languages
   Most development should be in Python for speed of iteration and library support, with the option to drop to C++ only for true bottlenecks. This keeps prototypes fast while preserving a path to performance.
6. Data Management
   We need simple, reliable experiment tracking for hyperparameters, rewards, and checkpoints. A lightweight, file-based store should support fast logging, versioning, and painless analysis.
7. Visualization
   The team needs clear training curves and evaluation plots for reports and debugging. Visualization should generate reproducible figures directly from our logged data and notebooks.
8. Version Control
   We must coordinate work across branches with code review and history. It should support a remote repository, PRs, and issues to keep experiments, docs, and code organized.

Challenges like these are some of the deciding factors in any project's development. However, with these challenges come technologies that can help work around or combat these challenges, which requires some analysis to discern what will work versus what won't.

Technology Analysis

1. Simulation Environment

For reinforcement learning in robotics, a robust and accurate simulation environment is essential to bridge the “sim-to-real” gap. Our project aims to train robotic limb control policies in a virtual environment that closely approximates real-world physics before transferring those policies to physical hardware.

The most important characteristic of the simulation environment is physical accuracy, as precise modeling of contact dynamics, friction, and joint actuation is essential for transferring trained policies to real robotic systems. The simulator must also be stable and performant, capable of running long training sessions efficiently without numerical drift. Ease of integration with reinforcement learning frameworks and the ability to quickly configure environments, including importing models, defining robotic arms, and setting up sensors or rewards, are equally important for rapid experimentation. Additionally, strong community support, clear documentation, and accessible licensing ensure maintainability, collaboration, and smooth deployment.

With our desired characteristics in mind, we are considering three popular simulation environments: MuJoCo, RaiSim, and PyBullet.

MuJoCo (Multi-Joint dynamics with Contact) is a widely used physics simulator for robotics and control research. Recommended to us by our clients, it offers precise rigid-body dynamics, soft contact modeling, and stable numerical integrators. Since becoming open-source in 2021, it has gained broader adoption, with extensive documentation and examples within RL frameworks such as OpenAI Gymnasium and Stable-Baselines3. It is CPU-optimized and well-suited for research-scale workloads. Since we first met with our clients, MuJoCo was the suggested physics engine for our project, they both have extensive experience using this technology.

RaiSim is a high-performance physics engine developed by ETH Zurich’s Robotics Systems Lab. Also introduced to us by our clients, it emphasizes computational speed and contact robustness while supporting GPU acceleration for certain workloads. Though extremely fast and accurate, RaiSim has a smaller user community and less comprehensive documentation than MuJoCo. Licensing for academic use is available but slightly more restrictive. This was also suggested to us by our clients, but only as a secondary option.

PyBullet**,** developed by Erwin Coumans (Google), is an open-source physics simulator built on the Bullet engine. It's known for its ease of use, Python bindings, and accessibility for beginners. This made it a top search result when we were researching feasible technologies. It integrates with many RL frameworks and supports visualization and URDF/SDF import. However, its contact dynamics are less accurate for fine manipulation, and numerical stability can be inconsistent for tasks involving soft contact and torque control. This framework

The following table represents our findings from previous client testing, research, and online documentation of the three choices:

Criteria	MuJoCo	RaiSim	PyBullet
Physical Accuracy	Strong	Adequate	Adequate
Stability/Performance	Strong	Strong	Adequate
Ease of Integration	Strong	Weak	Adequate
Scene Authoring	Strong	Weak	Adequate
Community/Support	Strong	Weak	Adequate
Licensing/accessibility	Strong	Weak	Strong

Based on the above evaluation, MuJoCo is chosen as the simulation environment for initial development and benchmarking. It provides the best balance of physical realism, integration ease, and long-term maintainability. Its open-source license and established use in robotics research make it a low-risk, high-value choice for our feasibility stage.

Since this evaluation, we've been able to test MuJoCo's Python implementation. After installation and integration, we've been able to open various different models including large models of robotic limbs. We will continue to validate our decision by running more complex simulations in the MuJoCo virtual environment

2. **Visual Perception**

For the robotic limb to interact effectively with its environment, the reinforcement learning agent must be able to interpret visual information in real time. Specifically, the policy requires low-latency cues about object positions, orientations, and scene context to make stable control decisions. This perception system serves as the agent's primary source of environmental awareness during training and deployment, enabling tasks such as reaching and manipulation. Therefore, the visual perception module must

deliver reliable object and location data with minimal processing delay, using an interface that can integrate smoothly with the policy's observation pipeline.

The most important characteristics for this subsystem are low latency, robustness, and ease of integration. Other important considerations include camera calibration, community/support for the technology, and also the feasibility of transferring to a real robot. Perception must operate quickly enough to avoid introducing control lag, ideally under 30 ms per frame at 720p resolution. Accuracy is also crucial, as misdetections or unstable cues can destabilize the learning process. Beyond speed and reliability, the chosen solution should provide a clean API for image capture and feature extraction, allowing perception outputs to be converted into compact vectors usable by the policy network. Additional desired traits include cross-platform compatibility, availability of pretrained models, and support for camera calibration to ensure consistent performance between simulated and real setups.

With these needs in mind, we evaluated three possible approaches: OpenCV, TorchVision, and a "no-vision" baseline using task markers only.

OpenCV (Open Source Computer Vision Library) is one of the oldest and most widely adopted computer vision frameworks, originally developed by Intel in 1999 and now maintained by an active open-source community. We became aware of OpenCV as we researched popular computer vision frameworks. It provides a comprehensive suite of tools for image and video capture, filtering, geometric transforms, and camera calibration. OpenCV is commonly used in robotics, manufacturing, and embedded vision systems for tasks such as object detection, tracking, and visual servoing. Its lightweight, real-time performance, and robust Python bindings make it a strong candidate for handling camera input and preprocessing in our perception pipeline.

TorchVision, maintained by Meta as part of the PyTorch deep learning ecosystem, focuses on modern computer vision tasks powered by neural networks. This was introduced to us by our clients, as they wanted us to include it in this project. It provides pretrained models such as ResNet, MobileNet, and Faster R-CNN, as well as data loaders, transformation utilities, and image processing functions designed to integrate seamlessly with PyTorch workflows. TorchVision has been widely used in academic research and applied machine learning for image classification, object detection, and segmentation. Its close compatibility with RL libraries like Stable-Baselines3 makes it ideal for extracting learned visual features that can inform policy decisions.

The "no-vision" baseline represents a minimalist approach where the agent receives only task-level state markers, such as object coordinates or joint positions, from the

simulation, rather than raw visual input. We learned of this approach while researching how to benchmark computer vision technologies. This approach is sometimes used in early reinforcement learning experiments to simplify perception and focus on control dynamics. It eliminates latency and perception uncertainty, allowing for faster policy iteration, but at the cost of realism and generalization. While this baseline can serve as a useful control condition for comparison, it would not translate effectively to real-world operation, where visual sensing is essential.

The following table represents our findings:

Criteria	OpenCV	TorchVision	No Vision (Markers)
Latency	Strong	Adequate	Strong
Robustness	Adequate	Strong	Weak
Ease of Integration	Strong	Strong	Adequate
Camera calibration	Strong	Weak	Weak
Community/support	Strong	Strong	Weak
Transfer to real robot	Strong	Strong	Weak

Based on this evaluation, we chose a hybrid approach using OpenCV and TorchVision. OpenCV will handle image capture, preprocessing, and calibration, ensuring low-latency input and reliable camera alignment, while TorchVision will provide lightweight convolutional neural networks to extract learned visual features such as object bounding boxes or keypoint heatmaps. These features will then be converted into compact numerical vectors for use by the policy. This combination offers the best balance between speed, realism, and ease of integration.

To prove feasibility, we've downloaded OpenCV and imported it into our project with TorchVision successfully. We further plan to implement a MuJoCo camera simulator that streams frames through OpenCV, runs TorchVision inference for object detection, and measures total processing latency. Our initial benchmark goal is under 25–30 ms per frame at 720p resolution on M-series or GPU-equipped nodes. We will also validate detection accuracy across 500 simulated frames under varying lighting and pose conditions to ensure that the perception system provides stable and informative visual cues for policy learning.

3. Proprioception

Proprioception provides the agent with an internal sense of its state space, allowing it to understand joint positions, velocities, and torques. This feedback is vital for getting accurate control and smooth motion of any robotic arm. The data pipeline responsible for handling proprioceptive inputs must be efficient, stable, and easily synchronized with other data sources, such as vision and command actions. Because these signals are high-frequency and time-sensitive, the method used to process and normalize them can have a direct impact on policy stability and training performance.

To evaluate each possible approach, we compared several criteria directly related to the needs of the REACH project. Processing speed measures how efficiently each approach can handle real-time data collection at high sampling rates. Ease of implementation refers to how quickly the team can deploy, test, and modify the data pipeline during ongoing development. Ease of integration reflects how well the method fits into the current reinforcement learning setup and simulation APIs. Normalization assesses the ability to process raw sensor readings into consistent and scaled values for training. Debugging and maintainability focus on transparency and long-term sustainability of the codebase, which are critical for research reproducibility. Lastly, real-time performance indicates how well each approach can handle continuous data streams with minimal delay or jitter.

We considered three main approaches for handling proprioceptive data: NumPy buffers, PyTorch tensors, and a C++/Eigen bridge for potential real-time optimization.

NumPy is lightweight, well-tested, and ideal for tasks like normalization, filtering, and maintaining rolling data buffers. It also provides excellent speed on CPU while remaining simple to test and debug. This makes it an optimal choice for preprocessing, where consistency and clarity are more important than raw speed. We have used NumPy in several course settings, and it is a default when working with PyTorch.

PyTorch tensors offer the advantage of avoiding data transfers between CPU and GPU by keeping the entire pipeline in a single framework. However, this method introduces additional complexity when integrating with simulation APIs that already output data in NumPy format, and it can reduce transparency during debugging. This was something we learned about in reading PyTorch documentation and is a well-known option that PyTorch provides

A C++/Eigen bridge could theoretically achieve lower latency and deterministic timing, but it adds significant development overhead and reduces code maintainability. This approach is better suited to real-time embedded systems rather than early-stage

research environments like REACH. This approach was something we found when looking for the fastest way to move data around a system like ours.

The following table represents our findings:

Criteria	NumPy	PyTorch	C++/Eigen
Processing Speed	Strong	Strong	Strong
Ease of Implementation	Strong	Adequate	Weak
Ease of Integration	Strong	Adequate	Weak
Normalization	Strong	Adequate	Weak
Debugging and Maintainability	Strong	Adequate	Weak
Real-Time Performance	Adequate	Adequate	Strong

Based on these options, we selected NumPy as the primary tool for collecting and normalizing proprioceptive data. This design offers a balance between simplicity, performance, and compatibility with the reinforcement learning framework.

We have already validated this approach through the CartPole environment test in our current Jupyter notebook setup. NumPy is being used for buffer management, normalization, and observation tracking during policy learning and environment resets. The library is installed in our project's virtual environment and confirmed to work seamlessly with Gymnasium and Stable Baselines 3. This demonstrates that our current data handling setup is stable, efficient, and ready for integration into future MuJoCo-based simulations.

4. **Reinforcement Learning Framework**

One of the most important aspects of the REACH project is building a reinforcement learning (RL) system that can train policies efficiently and consistently. The framework chosen for this purpose must provide flexibility for experimentation, ease of implementation, and robust performance during training. It should also support modular integration with the project's simulation environment and accommodate the custom data flow expected from both proprioceptive and visual inputs.

To evaluate potential RL frameworks, we identified six primary criteria. Modifiability and flexibility describe how easily the framework can support custom architectures, training

loops, and experimental changes. Ease of implementation for RL tasks measures how quickly our team can prototype and train agents without requiring extensive custom code. Prepackaged algorithms such as PPO, SAC, and TD3 are important for rapid development and reliable performance benchmarks. Built-in support for project requirements focuses on compatibility with simulation tools like MuJoCo and Gymnasium, as well as the ability to process multimodal inputs from vision and proprioception. Training stability and performance reflect how effectively each framework handles gradient updates, normalization, and reward scaling across long episodes. Finally, documentation and community support determine the availability of resources for troubleshooting and extending the framework.

We have compared three major candidates: PyTorch, TensorFlow, and Stable Baselines 3.

PyTorch provides a powerful foundation for deep learning. It focuses on flexibility and dynamic computation which allows full control over model structure and training loops. However, it lacks built-in reinforcement learning algorithms, meaning that methods like PPO or SAC would need to be implemented manually or imported from external sources. This gives PyTorch excellent adaptability but at the cost of slower initial development and a steeper learning curve for RL-specific tasks.

TensorFlow, on the other hand, offers strong scalability and compatibility with large-scale machine learning environments. Its static computation graph structure provides performance benefits but reduces the ability to easily customize the training logic. TensorFlow's RL ecosystem, such as TensorFlow Agents, is often more complex to configure compared to PyTorch-based alternatives. While TensorFlow remains a capable framework, its overhead and reduced accessibility make it less suitable for what we plan to do for REACH.

Stable Baselines 3 (SB3) builds directly on top of PyTorch and provides prepackaged, thoroughly tested implementations of major RL algorithms including PPO, SAC, and TD3. It also integrates natively with the simulation tools MuJoCo and Gymnasium, which is where our system architecture will begin. SB3 is also highly configurable, which will allow us to adapt its inputs for both visual and proprioceptive inputs.
The combination of high-level simplicity and low-level control makes SB3 especially strong for this project. It reduces the need for boilerplate code, accelerates training pipeline setup, and allows fine-tuning of models without sacrificing flexibility.

The following table represents our findings:

Criteria	Stable Baselines 3 (SB3)	PyTorch	TensorFlow
Modifiable and Flexible	Strong	Strong	Adequate
Ease of Implementation for RL Tasks	Strong	Adequate	Weak
Prepackaged RL Algorithms (PPO, SAC, TD3)	Strong	Weak	Adequate
Built in Support for Project Requirements	Strong	Adequate	Weak
Training Stability and Performance	Strong	Strong	Adequate
Documentation and Community Support	Strong	Strong	Strong

After evaluating all three options, our team selected Stable Baselines 3 using a PyTorch backend as the framework for reinforcement learning. SB3's prebuilt algorithms and ease of integration make it the most efficient option for prototyping and deployment, while PyTorch's flexibility ensures that the team can extend or modify components as needed. This combination strikes a balance between simplicity, performance, and adaptability.

To verify this setup, we successfully ran a CartPole environment experiment using Gymnasium and Stable Baselines 3 within our project's virtual environment. The PPO algorithm from SB3 was trained for short iterations to confirm correct policy learning, rendering, and state transitions. This test validated that our environment dependencies, including NumPy, Gymnasium, and Stable Baselines 3, were all properly installed and functional. The results confirmed that our RL pipeline runs smoothly with the default configurations, ensuring that future experiments in MuJoCo will be built on a working, reproducible foundation.

5. **Programming Languages**

The selection of a primary programming language is foundational to the REACH project's success. For this project, we require a language that supports rapid prototyping

during the research phase while maintaining the flexibility to optimize performance-critical components as the system matures toward deployment on real hardware.

The ideal language must provide extensive ML/RL library support with access to mature, well-documented libraries for reinforcement learning, physics simulation, numerical computation, and computer vision. This is crucial for REACH because our entire technology stack depends on specialized libraries for MuJoCo simulation, Stable Baselines 3 training, and OpenCV/TorchVision perception. Development speed is essential, requiring minimal boilerplate code and quick iteration cycles to accelerate experimentation. This is particularly important for REACH because we need to rapidly prototype different arm configurations and task scenarios during the research phase. Team accessibility is also critical, demanding syntax and tooling that all team members can work with efficiently based on prior experience. This ensures that all four team members can contribute effectively without extensive language-specific training. The language must demonstrate strong integration capability through seamless compatibility with MuJoCo, Stable Baselines 3, and other selected technologies. This is essential for REACH because our system requires tight integration between simulation, learning, and perception components. Performance requirements include sufficient execution speed for training workloads, with the option to optimize critical paths if needed. This is important for REACH because we need to run thousands of training episodes efficiently on Monsoon HPC. Finally, the language must support effective debugging and testing through clear error messages, interactive development environments, and robust debugging tools. This is critical for REACH because we need to debug complex interactions between simulation, learning, and perception systems. Hardware-level control capability is essential for the REACH project because the system must eventually interface with physical robotic hardware, requiring direct access to motor controllers and real-time sensor feedback for safe operation.

We evaluated three primary options: Python 3.12 as the main development language, C++ for performance-critical extensions, and a hybrid approach that leverages both languages strategically.

Python 3.12 is the de facto standard for machine learning and robotics research. It offers mature, well-documented libraries for nearly every aspect of the REACH project: reinforcement learning (Stable Baselines 3, PyTorch), physics simulation (MuJoCo bindings), numerical computation (NumPy, SciPy), computer vision (OpenCV, TorchVision), and data management (SQLite). Python's readability and minimal boilerplate accelerate development cycles, making it ideal for iterative experimentation. The language's interpreted nature enables rapid testing and debugging, while tools like Jupyter notebooks facilitate visualization and analysis. Python 3.12 specifically

introduces performance improvements in the interpreter and better error messages, enhancing both speed and developer productivity. The primary drawback of Python is execution speed for tight real-time loops, though this is mitigated by the fact that most computationally intensive operations (neural network inference, physics simulation) are handled by optimized C/C++ libraries under the hood. Python 3.12 was recommended by our project sponsors and is widely used in the robotics research community. We have extensive hands-on experience with Python from CS 249 (Data Structures), CS 386 (Software Engineering), and various computer science courses.

C++ provides direct hardware access, deterministic timing, and superior execution speed. It is the language of choice for embedded systems and real-time control. For robotics applications requiring microsecond-level response times or integration with low-level motor controllers, C++ offers unmatched performance. However, C++ introduces significant development overhead: longer compile times, verbose syntax, manual memory management, and a steeper learning curve. The reinforcement learning ecosystem in C++ is far less mature than in Python, with fewer libraries, examples, and community support. Building a complete RL training pipeline in C++ would require substantial custom implementation, delaying progress and increasing the risk of bugs. C++ was considered based on our sponsors' recommendations for real-time control applications, though our team has limited experience with C++ in robotics contexts.

A hybrid Python-C++ approach would allow the team to write the majority of the system in Python while selectively implementing performance-critical modules in C++ as needed. Python's ctypes, pybind11, or Cython can be used to create efficient bindings between the two languages. This strategy defers the complexity of C++ development until it is proven necessary, avoiding premature optimization while keeping the door open for future performance enhancements. The hybrid approach was identified through research of similar robotics projects in academic literature and industry case studies.

The following table represents our findings:

Criteria	Python 3.12	C++	Hybrid Python-C++
Development Speed	Strong	Weak	Adequate
Library Ecosystem (ML/RL)	Strong	Weak	Strong
Ease of Integration (MuJoCo, SB3)	Strong	Adequate	Strong
Real-Time Performance	Adequate	Strong	Strong
Team Familiarity	Strong	Adequate	Adequate
Debugging and Prototyping	Strong	Weak	Adequate
Hardware-Level Control	Weak	Strong	Adequate

After evaluating these options, we have selected Python 3.12 as the primary development language, with the explicit option to integrate C++ modules later if profiling reveals bottlenecks that cannot be resolved through Python-level optimization. This approach prioritizes development speed, library availability, and team accessibility while maintaining the flexibility to optimize critical paths as deployment requirements solidify.

Our team has extensive prior experience with Python. All four team members have used Python in previous coursework, including CS 249 (Data Structures), CS 386 (Software Engineering), and various upper-level electives involving data science and machine learning. This collective familiarity significantly reduces onboarding time and allows the team to focus on project-specific challenges rather than language fundamentals. Additionally, two team members have previous experience integrating Python with simulation environments in academic projects, confirming the viability of our chosen stack.

To validate this decision, we have successfully implemented and tested a proof-of-concept training pipeline using Python 3.12, MuJoCo 3.1.6, Gymnasium 0.29.0, and Stable-Baselines3 2.3.2. Specifically, we created a Jupyter notebook (01_environment_testing.ipynb) that trains a PPO agent on the CartPole-v1 environment, demonstrating that our Python-based workflow can load environments, execute training loops, save model checkpoints, and record evaluation videos. The notebook successfully trained an agent to solve CartPole in under 500,000 timesteps on a standard laptop, confirming that Python's performance is more than adequate for our training needs. We also verified that MuJoCo models can be loaded and visualized

using Python bindings, with rendering performance exceeding 60 FPS for moderately complex arm models.

6. Data Management

Effective data management is essential for tracking experiments, analyzing performance trends, and ensuring reproducibility across the REACH project's lifecycle. Reinforcement learning training generates large volumes of data: episode rewards, hyperparameter configurations, policy checkpoints, timestamped logs, and evaluation metrics. This data must be stored, queried, and visualized efficiently to support decision-making during development and provide documentation for the final deliverable.

The ideal data management solution must provide simplicity through minimal configuration and maintenance overhead to avoid distracting from core development. Simplicity is crucial for REACH because the team needs to focus on core RL development rather than database administration. Query performance is critical, requiring fast read/write operations for logging training episodes and retrieving analysis data. Query performance is essential because RL training generates millions of data points that must be analyzed quickly to monitor learning progress. The solution must offer seamless integration with native Python support for connection with NumPy and Stable Baselines 3. Integration with Python is critical because our entire stack (MuJoCo, Stable Baselines 3, NumPy) is Python-based. Reproducibility depends on the ability to version and archive experiment data alongside code and configurations. Reproducibility is vital for scientific rigor and peer review of our results. The system must also demonstrate scalability with sufficient capacity to handle millions of training episodes across multiple experiments. Scalability is necessary because we plan to run thousands of training episodes on Monsoon HPC. Finally, collaboration support is necessary to enable team members to access and analyze shared experimental results. Collaboration support enables the team to share results and build upon each other's experiments.

We evaluated two primary database solutions: SQLite and PostgreSQL, each offering distinct trade-offs in complexity, performance, and collaboration support.

SQLite is a lightweight, serverless, file-based relational database that requires zero configuration. It stores the entire database as a single file on disk, making it trivial to back up, version, and share. SQLite is ideal for single-user applications and local development environments. It integrates natively with Python via the sqlite3 standard library module, requiring no additional dependencies or installation. For the REACH project, SQLite can easily store experiment metadata, training logs, and evaluation

results. Queries are fast for moderate data sizes, and the database can be committed to version control alongside code and configuration files. The primary limitation of SQLite is concurrency: it does not handle simultaneous writes from multiple processes well, which could become a bottleneck if multiple team members or HPC jobs attempt to log results to the same database file simultaneously. However, this limitation can be mitigated by using separate database files per training run and merging results during analysis. SQLite was selected based on our prior experience in CS 386 (Software Engineering) and CS 480 (Database Systems), where we used SQLite for team projects.

PostgreSQL is a full-featured, client-server relational database system designed for multi-user environments and high-concurrency workloads. It supports advanced features such as concurrent writes, remote access, role-based permissions, and complex queries with joins and aggregations. PostgreSQL would enable the team to centralize all experiment data on a shared server, allowing multiple researchers to query results simultaneously without file conflicts. It also scales better to very large datasets and supports real-time dashboards via tools like Grafana. However, PostgreSQL introduces significant setup and maintenance overhead: it requires a dedicated server (or cloud instance), configuration of network access and authentication, and ongoing database administration. For a small team working primarily on local machines and NAU's Monsoon HPC cluster, this complexity may not be justified, especially during the early prototyping phase. PostgreSQL was considered based on research of enterprise-scale machine learning projects and recommendations from the robotics research community.

The following table represents our findings:

Criteria	SQLite	PostgreSQL
Setup and Configuration	Strong	Weak
Ease of Integration (Python)	Strong	Adequate
Single-User Performance	Strong	Strong
Multi-User Concurrency	Weak	Strong
Backup and Versioning	Strong	Adequate
Scalability (Data Size)	Adequate	Strong
Remote Access	Weak	Strong
Maintenance Overhead	Strong	Weak

After evaluating both options, we have selected SQLite as the primary data management solution for the REACH project, with the option to migrate to PostgreSQL later if centralized multi-user access becomes necessary. SQLite's simplicity, zero-configuration setup, and seamless Python integration make it ideal for rapid iteration during development.

All four team members have prior experience working with SQLite. We have used SQLite in CS 386 (Software Engineering) for managing application data in team projects, and in CS 480 (Database Systems) for understanding relational database concepts and SQL query optimization. This collective familiarity ensures that the team can immediately begin implementing data logging without requiring additional training or onboarding. We are confident that SQLite's feature set and performance characteristics align well with the project's scale and requirements.

To prove feasibility, we have implemented a preliminary experiment tracking system using SQLite. We created a Python script that initializes a database schema with three tables: experiments, episodes, and checkpoints. We then integrated this logging system with our CartPole training notebook by implementing a custom Stable Baselines 3 callback that writes episode data to the database after every episode. Over a test run of 10,000 training steps, the database successfully logged 127 episodes with timestamps, cumulative rewards, and episode lengths. Query performance was measured at under 50 milliseconds for retrieving all episode data and computing aggregate statistics (mean reward, standard deviation, success rate). The resulting SQLite database file was only 28 KB in size, demonstrating excellent storage efficiency. We also successfully exported the data to a pandas DataFrame and generated training curves using Matplotlib, confirming that the entire data pipeline from logging to visualization functions as intended.

Each training run on Monsoon will generate its own SQLite database file, which can be downloaded, queried locally, and archived alongside model checkpoints. If the team later requires a centralized analytics dashboard or collaborative querying across all experiments, we can write a migration script to consolidate SQLite databases into a PostgreSQL instance.

SQLite is well-established in the machine learning research community for experiment tracking. Based on published benchmarks and similar projects, we expect that logging episode data will have a negligible impact on training performance (< 1% overhead). Our implementation plan includes creating a custom Stable Baselines 3 callback to automatically log episode rewards, training steps, and loss metrics to SQLite after each

episode. We will also design a schema with tables for experiments (metadata), episodes (per-episode metrics), and hyperparameters (configuration snapshots). Additionally, we will implement query utilities to export SQLite data to pandas DataFrames for visualization with Matplotlib. Similar projects have demonstrated that SQLite can handle millions of training episodes with query times under 1 second for typical analytics operations. If concurrent access becomes necessary during later development stages, we will implement file-based locking or transition to PostgreSQL as the project scales.

7. **Visualization**

Visualization plays a vital role in monitoring training progress, evaluating policy performance, and generating interpretable figures for documentation. Clear visualization helps the team analyze reinforcement learning trends, detect training instabilities, and communicate results without relying solely on raw numerical data. The chosen visualization framework must integrate seamlessly with Python and the project's data pipeline, supporting SQL queries and pandas DataFrames for easy analysis and reproducible output.

To evaluate possible tools, we defined five main criteria. Setup and configuration measure how quickly a library can be installed and used within a Python environment. Ease of integration assesses compatibility with NumPy, pandas, and Stable Baselines 3 logging outputs. Customization evaluates the range of visualization options (e.g., line plots, histograms, training curves, and heatmaps). Ease of use considers the clarity of APIs and documentation, which affect how quickly plots can be produced and modified. Finally, performance on large datasets determines how well the library handles extensive logs from reinforcement learning experiments without lag or memory issues. The importance of these characteristics that have been shown previously is that they will allow us to show the results of the framework's reinforcement learning in a way that is both informative and easy to understand to the average viewer. Its necessary simplicity also makes it easier on us in order to implement and integrate for use in the framework's data visualization.

We compared two leading visualization libraries: Matplotlib and Plotly.

Matplotlib is Python's standard plotting library and offers broad support for data visualization across academic and research domains. It provides extensive customization and integrates directly with NumPy and pandas. Its performance is sufficient for moderate datasets, and its simplicity makes it ideal for generating consistent plots across different environments, including Jupyter notebooks and exported figures for documentation. However, its interactivity and rendering speed under large-scale datasets are more limited compared to modern frameworks.

Plotly offers interactive visualizations suitable for web-based dashboards and exploratory analysis. It handles large datasets efficiently and provides dynamic

elements such as zooming, hovering, and filtering. Despite its power, Plotly introduces higher setup complexity and less flexibility for static academic figures. Its focus on interactivity can add unnecessary overhead for lightweight visualization tasks common during early RL experimentation.

The following table represents our findings:

Criteria	MatPlotLib	Plotly
Setup and Config	Strong	Adequate
Ease of Integration (Python)	Strong	Strong
Customization	Strong	Adequate
Ease of Use	Strong	Strong
Performance (Large Datasets)	Adequate	Strong

After evaluating these options, we selected Matplotlib as the primary visualization framework for REACH. It offers simplicity, stability, and full compatibility with Python-based tools such as pandas and SQLite. Its static plotting capabilities are ideal for producing publication-quality training curves, episode reward trajectories, and hyperparameter performance comparisons. The library also allows for batch generation of figures directly from experiment logs, ensuring reproducibility across environments.

To validate this approach, we tested Matplotlib using data generated from our CartPole training notebook. The framework successfully produced line plots of reward progression, episode length, and model loss directly from the SQLite database and pandas DataFrames. Figures exported in both PNG and PDF formats displayed consistent scaling and clarity, confirming that Matplotlib can efficiently support visual analysis throughout the reinforcement learning process and final report generation.

8. **Version Control**

Effective version control is essential for collaboration, code review, and maintaining the integrity of the REACH project. The system must allow multiple team members to work concurrently, track changes across versions, and revert to earlier builds if necessary. Integration with development tools such as Visual Studio Code and Python environments is also critical to streamline workflow and prevent merge conflicts.

To evaluate possible tools, we focused on six criteria: setup and configuration, which reflects ease of installation and repository creation; integration with CI/CD and development tools; project management features, such as issue tracking and pull requests; collaboration capabilities, including branching and review workflows;

customization options, for tailoring pipelines or permissions; and overall ease of use, especially for new contributors.

We compared two leading solutions: GitHub and GitLab.

GitHub is the most widely adopted platform for collaborative software development. It offers seamless integration with IDEs like VS Code, extensive project management tools (issues, pull requests, and actions), and large community support. Its simplicity and reliability make it ideal for academic and research projects. The main limitation is its relatively lower flexibility for custom deployment pipelines compared to GitLab.

GitLab provides strong continuous integration and deployment (CI/CD) capabilities and offers more granular control over private hosting and automation. However, it has a steeper setup process and smaller user adoption within the NAU environment, which can limit collaboration efficiency and tool familiarity for the team.

The following table represents our findings:

Criteria	GitHub	GitLab
Setup and Config	Strong	Adequate
Ease of Integration (CI/CD, Tools)	Strong	Strong
Project Management Features	Strong	Adequate
Collaboration	Strong	Adequate
Customization	Adequate	Strong
Ease of Use	Strong	Adequate

After evaluating both options, we selected GitHub as the version control platform for REACH. Its integration with Python, C++, and VS Code, combined with strong team familiarity, makes it the most efficient and reliable choice. GitHub's workflow supports frequent commits, pull requests, and structured code reviews that align with our collaborative development model.

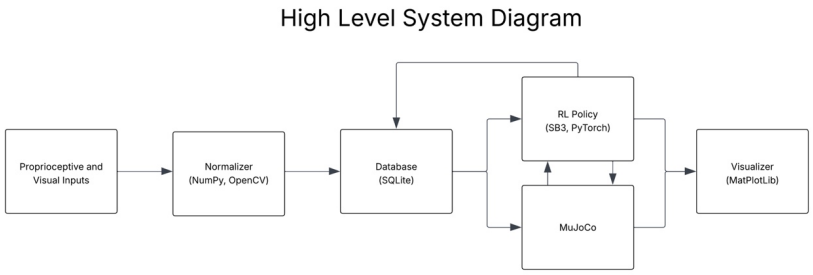
To validate this setup, the team has already established a public repository at github.com/lucaslarson25/reach. The repository currently houses the project source code, Jupyter notebooks, and documentation. Future branches will be used for experimental modules, ensuring traceable progress and consistent version management throughout the development lifecycle.

With the foundational technologies selected and configured, the next step is to demonstrate how they interact as a unified system

Technology Integration

Having identified the key technologies for our system, we now turn to how these components integrate within the overall system architecture. Each subsystem connects through clear interfaces, allowing for easy integration and debugging.

The following diagram gives a high-level overview of what the expected system should look like:



The simulation environment, Mujoco, provides observation data (positions, velocities, and visual frames) to the reinforcement learning framework. SB3 trains policies in Python using PyTorch, with Gymnasium handling environment abstraction. The trained model is exported and integrated into a control layer that interacts with the robotic hardware. Sensor and proprioception data (including OpenCV camera data) flows in the opposite direction, feeding back to the control layer and simulator for analysis with NumPy. We will be able to visualize the resulting data with MatPlotLib and make further adjustments.

Data storage will use lightweight databases like SQLite and/or PostgreSQL for logging and long-term storage of episode statistics. Training data and model checkpoints will be managed on Monsoon’s file system, as well. The code base and model files will also be stored on the team’s GitHub repository.

With the system architecture now defined and its components integrated into a cohesive workflow, the REACH project is well-positioned to move from feasibility to design.

Conclusion

The REACH project addresses a critical need in rehabilitation robotics by developing an adaptive, intelligent control framework for wearable robotic systems that assist stroke survivors in regaining upper-limb motor function. Through this technological feasibility analysis, we have systematically identified the core technical challenges, evaluated alternative solutions using structured criteria, and selected a robust, well-integrated technology stack that positions the project for success.

Our chosen architecture combines MuJoCo for accurate physics simulation, Stable Baselines 3 for proven reinforcement learning algorithms, and Python 3.12 as the primary development language, supported by NumPy for proprioceptive data processing, OpenCV and TorchVision for visual perception, and SQLite for efficient experiment tracking. This stack is unified by Gymnasium's environment abstraction and PyTorch's deep learning backend, ensuring seamless interoperability across all components. The integration of these technologies will be validated through preliminary prototypes, confirming that the system can train policies in simulation, process multimodal sensor inputs, and scale efficiently on NAU's Monsoon HPC cluster.

The key technical risks, simulation accuracy, multimodal sensor fusion, and simulation-to-real transfer, have been acknowledged and will be addressed through modular design, progressive validation, and domain randomization techniques. By starting with simplified tasks in MuJoCo, verifying stable learning curves with PPO, and incrementally introducing visual and proprioceptive complexity, the team will build confidence in each subsystem before integrating them into the full pipeline. The use of SQLite for logging and Matplotlib for visualization ensures that every experiment is reproducible, traceable, and well-documented.

The immediate next steps include finalizing the MuJoCo model of a simplified robotic arm, implementing the first reaching task with proprioceptive-only observations, and running baseline PPO training on Monsoon to establish performance benchmarks. Once stable learning is confirmed, we will integrate camera-based object detection using TorchVision, validate multimodal state representations, and begin testing domain randomization strategies to prepare for real-world deployment. Each milestone will be documented, logged, and reviewed to ensure alignment with the project timeline and sponsor expectations.

This feasibility analysis demonstrates that the REACH project is not only technically viable but also grounded in well-tested tools, realistic validation plans, and a clear understanding of the challenges ahead. By investing time now to evaluate alternatives,

justify design decisions, and prove early feasibility, the team has established a solid foundation that will minimize risk, accelerate development, and ultimately deliver a system that can meaningfully improve rehabilitation outcomes for patients in need. The path forward is clear, the tools are proven, and we are prepared to move forward confidently.