Design Document: Reinforcement Learning for the Racetrack Problem

Taron Moe-Stull
taronmoe@icloud.com

November 2025

System Requirements

This document describes the design of a reinforcement learning system applied to the racetrack problem. The system is designed to run end-to-end inside a Jupyter notebook, driven from a single parameter cell so that a user can swap tracks, learning algorithms, and crash handling rules without modifying the code.

The core goal is to implement three control methods from scratch—Value Iteration, Q-Learning, and SARSA— then train and evaluate on multiple tracks and generate both a trajectory visualization and learning-curve metrics.

Configuration Parameters

The notebook is driven by the following parameters:

- **GROUP_ID**: string identifier used in output filenames (can be treated as a project tag in portfolio usage)
- **ALGORITHM**: one of `ValItr`, `QLrng`, or `SARSA`
- **TRACK_NAME**: path to a `.txt` track file (e.g., `/tracks/U-track.txt`)
- **CRASH_POS**: crash handling mode, either `NRST` (nearest valid cell) or `STRT` (restart)

These fields are validated before execution. The system reads directly from this configuration, and no algorithm names, file paths, or crash settings are hard-coded.

Track Input Format

Each racetrack is provided as a plain text file:

- First line: grid dimensions in `rows,cols` format
- Remaining lines: ASCII grid using symbols:
  - `S` start cell
  - `F` finish cell
  - `.` open track
  - `#` wall

The system must validate that the dimensions match the header and locate all start and finish cells. Because the car can move more than one cell per step, collision detection must evaluate the *entire line segment* between positions, not just the endpoint.

Environment Dynamics

A state consists of position and velocity: (x, y) and (v_x, v_y), with velocity bounded to $[-5, 5]$. At each step, the agent chooses acceleration (a_x, a_y) where each component is in $\{-1, 0, 1\}$.

With 80% probability the acceleration applies normally; with 20% probability the acceleration fails and is replaced by $(0, 0)$. Velocity is updated subject to bounds. Each step incurs a cost of 1, except for the move into the finish cell, which ends the episode with 0 cost for that final move.

Outputs

After training, the system must:

- generate a plot visualizing a sample trajectory produced by the learned policy
- run repeated experiments per track (minimum 10) to produce learning curve statistics for analysis

The required visualization filename follows:

`[GROUP_ID]_[ALGORITHM]_[TRACK_NAME]_[CRASH_POS].png`

System Architecture

The system is organized as a modular pipeline driven by the notebook runner. It begins with parameter loading, parses the track file into a structured model, constructs an environment with the correct dynamics, trains a policy using the selected algorithm, runs repeated experiments to produce learning curves, and finally generates the required path visualization.

Notebook (Top-Level Runner)

Role: Executes the full pipeline: parameters → track → environment → algorithm → experiments → plots.
Inputs: `GROUP_ID`, `ALGORITHM`, `TRACK_NAME`, `CRASH_POS`.
Outputs: Calls to `TrackLoader`, `RacetrackEnv`, selected RL algorithm, `ExperimentManager`, and `Plotter`; produces experiment metrics and PNG output.
Major Methods: `run()`, `load_params()`, `init_track()`, `init_env()`, `select_algorithm()`, `run_experiments()`, `generate_plots()`.

Parameters

Role: Container for configuration values.
Inputs: Parameter cell values.
Outputs: Validated configuration for `TrackLoader`, `RLAlgorithms`, and `Plotter`.
Major Methods: `validate()`.

TrackLoader

Role: Reads and validates the racetrack `.txt` file.
Inputs: `TRACK_NAME` path; header and ASCII grid.
Outputs: A `TrackModel` containing dimensions, grid, start cells, finish cells.
Major Methods: `load_track(path)`, `parse_header()`, `parse_grid()`, `validate_symbols()`.

TrackModel

Role: The system representation of the track used by `RacetrackEnv` and plotting utilities.
Inputs: Parsed data from `TrackLoader`.
Outputs: Query functions for walls, finish cells, valid coordinates, start/finish sets.
Major Methods: `cell_at()`, `is_wall()`, `is_finish()`, `start_positions()`, `finish_positions()`.

RacetrackEnv

Role: Simulates motion, acceleration failure, velocity bounds, crash handling, and termination.
Inputs: `TrackModel`, `CRASH_POS`, current state $(cell, velocity)$, action (a_x, a_y).
Outputs: `next_state`, `cost`, `done`.
Major Methods: `reset()`, `step(action)`, line-tracing collision helper, nearest-valid-cell helper, crash handler.

RLAlgorithms

Role: Selects and trains the requested algorithm.
Inputs: `ALGORITHM`, `RacetrackEnv`, algorithm-specific parameters.
Outputs: Policy representation (state $\rightarrow$ action, or Q-values) and training statistics.
Major Methods: `select_algorithm()`, `train_value_iteration()`, `train_q_learning()`, `train_sarsa()`.

ExperimentManager

Role: Runs repeated training/evaluation trials to produce learning curve data.
Inputs: Environment, algorithm selection, experiment configuration.
Outputs: Performance metrics (e.g., steps-to-finish per run).
Major Methods: `run_experiments()`, `evaluate_policy()`, averaging/logging helpers.

Plotter

Role: Generates the required PNG figure showing a sample race path.
Inputs: `TrackModel`, evaluated trajectory, parameters.
Outputs: Saved figure named `[GROUP_ID]_[ALGORITHM]_[TRACK_NAME]_[CRASH_POS].png`.
Major Methods: `plot_track_with_path()`, grid drawing utilities, path overlay.

Architecture Diagram

Upload the architecture diagram extracted from the original PDF (page 3) as: `racetrack_arch.png`. :contentReference[oaicite:2]index=2

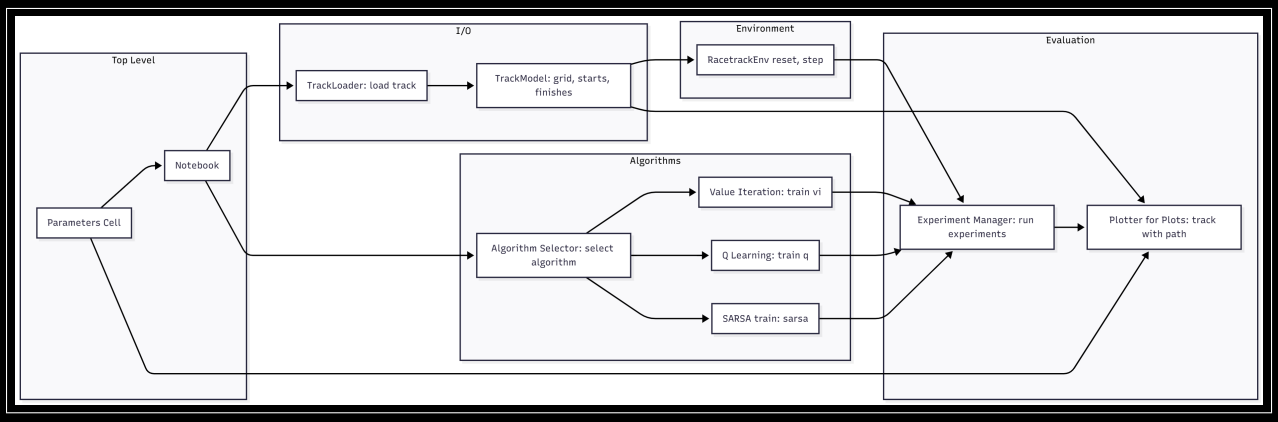

Figure 1: System pipeline and module-level architecture for the racetrack reinforcement learning system.

System Flow

The system operates as a single linear pipeline beginning with the parameters cell and ending with the required trajectory PNG and learning curve data.

1. **Parameter Loading:** Read and validate `GROUP_ID`, `ALGORITHM`, `TRACK_NAME`, and `CRASH_POS`. Ensure the selected algorithm is one of `ValItr`, `QLrng`, `SARSA`, the track points to a valid `.txt` file, and the crash rule is either `NRST` or `STRT`.
2. **Track Parsing:** Call `TrackLoader` to read the ASCII track file, parse the header and grid, identify start and finish cells, validate symbols, and construct a `TrackModel`.
3. **Environment Initialization:** Construct a `RacetrackEnv` using `TrackModel` and `CRASH_POS`. Initialize the start position and velocity $(0, 0)$. Enforce acceleration failure (80/20), velocity bounds, line-segment collision detection, and crash behavior.
4. **Algorithm Selection:** Select the trainer from `RLAlgorithms` based on `ALGORITHM`.
5. **Value Iteration:** If selected, construct a value table over the state space using environment transition rules, apply iterative value updates until convergence, then extract a greedy policy from the final values.
6. **Q-Learning / SARSA:** If selected, initialize a Q-table and interact with the environment for multiple episodes. Select actions, observe transitions via `step(action)`, update Q-values using the chosen update rule, and produce an improved policy.
7. **Experiment Management:** Run repeated training/evaluation cycles (minimum 10 per track) to produce learning curve data such as steps-to-finish.

8. **Policy Evaluation and Trajectory:** Evaluate the final policy once to produce a sample trajectory used for visualization.
9. **Output Generation:** Send the trajectory, `TrackModel`, and parameters to the plotter, render the track and path overlay, and save `[GROUP_ID]_[ALGORITHM]_[TRACK_NAME]_[CRASH_POS].png`.
10. **Metrics Collection:** Store run counts, training curves, and aggregate steps-to-finish to support analysis of algorithm behavior and crash mode effects.

Test Strategy

Testing is structured around acceptance criteria from parameter handling through plotting.

First, validate configuration input: the system must accept only supported algorithm names, valid track paths, and valid crash modes. Next, validate track parsing: ensure dimensions, symbols, start cells, and finish cells match the track file exactly.

Then validate environment mechanics: velocity bounds, the 80/20 acceleration rule, collision detection using line tracing, both crash modes (`NRST`, `STRT`), and termination on reaching finish with 0 cost for the final move.

Algorithm testing proceeds from simple to complex:

- **Value Iteration:** verify convergence on simpler tracks and that the resulting policy reaches a finish state.
- **Q-Learning / SARSA:** verify improvement trends over training (e.g., decreasing steps-to-finish over time).

Finally, output verification checks that:

- the saved PNG uses the required filename format
- the plotted trajectory corresponds to a valid sequence consistent with the learned policy and environment rules

Roadmap

The system can be implemented and validated in stages:

1. Finalize parameter validation and notebook runner scaffolding.
2. Implement and validate `TrackLoader` and `TrackModel` across all tracks.
3. Implement `RacetrackEnv` mechanics: movement, bounds, collision tracing, crash modes, termination.
4. Implement training for Value Iteration, Q-Learning, and SARSA with a consistent policy interface.
5. Integrate `ExperimentManager` for repeated trials and learning curve logging.
6. Implement `Plotter` and confirm required naming and path overlay output.

7. Run end-to-end experiments across tracks and crash modes and validate outputs.

References

1. GeeksforGeeks. 2023. "Q-Learning in Reinforcement Learning." https://www.geeksforgeeks.org/q-learnin
2. GeeksforGeeks. 2023. "SARSA Reinforcement Learning Algorithm." https://www.geeksforgeeks.org/sarsa
3. GeeksforGeeks. 2023. "Reinforcement Learning – A Complete Guide." https://www.geeksforgeeks.org/what
4. Mermaid Project. 2024. "Mermaid Live Editor." https://mermaid.live/
5. Russell, S., and P. Norvig. 2020. *Artificial Intelligence: A Modern Approach.* 4th Edition. Pearson Higher Education.