Think Before Act

Neural Surrogate World Models for LLM-Guided MPC

Research note

Markus Buchholz

Abstract

This research report documents a practical *Model Predictive Control* (MPC) architecture applied to the MiniWorld Maze navigation task [1]. The central idea is simple but powerful: rather than acting greedily on the current observation, the agent *thinks before it acts* - it simulates a small number of candidate futures inside a learned **world model** [2], scores each predicted outcome, and executes only the first action of the best-scoring plan before replanning (receding-horizon control). This direction is consistent with recent advances in world-model research that emphasise internal simulation and learned dynamics as foundations for planning and decision making [3].

The first part of the note develops the conceptual framework: state representation, dynamics, an image-derived objective function, and the receding-horizon MPC optimisation problem, all illustrated on MiniWorld. The second part presents a complete neural surrogate pipeline that *replaces* the expensive environment simulator with a lightweight convolutional world model trained on real transitions. At inference time the planner queries only this neural surrogate; the real environment is invoked solely for committed actions. On MiniWorld we achieve a 7.02× speedup.

The proposed optimiser belongs to the same family as *Model Predictive Path Integral* (MPPI) control [4]: both sample K candidate trajectories, roll them forward through a dynamics model, score each rollout, and commit only the first action of the best plan. The key difference is that MPPI draws trajectories by random Gaussian perturbation and aggregates them via an information-theoretic softmax weighting, whereas our approach generates a small number of semantically meaningful candidates-the discrete actions available in MiniWorld-and selects among them by hard argmax. This makes the method well-suited to settings where intelligent candidate generation can replace statistical coverage.

In addition, the proposed architecture is compatible with emerging *latent-action world models* that learn abstract, continuous action spaces directly from large collections of in-the-wild videos without action labels [5]. Because our MPC optimiser operates entirely over latent dynamics, it can naturally consume such learned action spaces, enabling embodiment-free planning and aligning with these state-of-the-art approaches to scalable, action-agnostic control.

Purpose of this work. MiniWorld is used here as a *proof-of-concept* reinforcement learning environment: a well-understood, open-source setting that lets us validate world-model learning, surrogate accuracy, and MPC inference speed end to end, without the complexity or cost of a proprietary, high-fidelity simulator.

1 Introduction

Large Language Models (LLM) are increasingly used as high-level planners for embodied agents. A natural and principled approach is MPC: before committing to an action, the agent simulates K candidate trajectories of length H in a *world model*, scores each outcome, and executes only the best first action.

For K=3 candidate first-actions and H=7 horizon steps, the greedy MPC kernel evaluates all $|\mathcal{A}|$=3 possible continuations at every step to select the best, giving a total per-planning-call cost of:

$$N_{\text{sim}} = K \times |\mathcal{A}| \times H = 3 \times 3 \times 7 = 63 \tag{1}$$

simulator calls. When the real environment is a physics engine, game engine, or robotic system, these 63 calls can take 100 ms–10 s per planning step, making real-time LLM-guided navigation completely infeasible.

The **surrogate approach** trains a neural network f_θ on previously collected transitions. Once trained, f_θ replaces the simulator: a full 63-call planning horizon costs ≈ 3.8 ms regardless of the original simulator's complexity. This note presents the full picture-from the conceptual MPC loop, through the neural surrogate architecture, to the measured speed-up this gives over planning directly inside the simulator.

The receding-horizon planning loop is illustrated in Figure 1: at time t the agent simulates a small set of candidate futures in the model world, scores them with an objective function, executes only the first action of the best-scoring plan, and replans at time t+1.

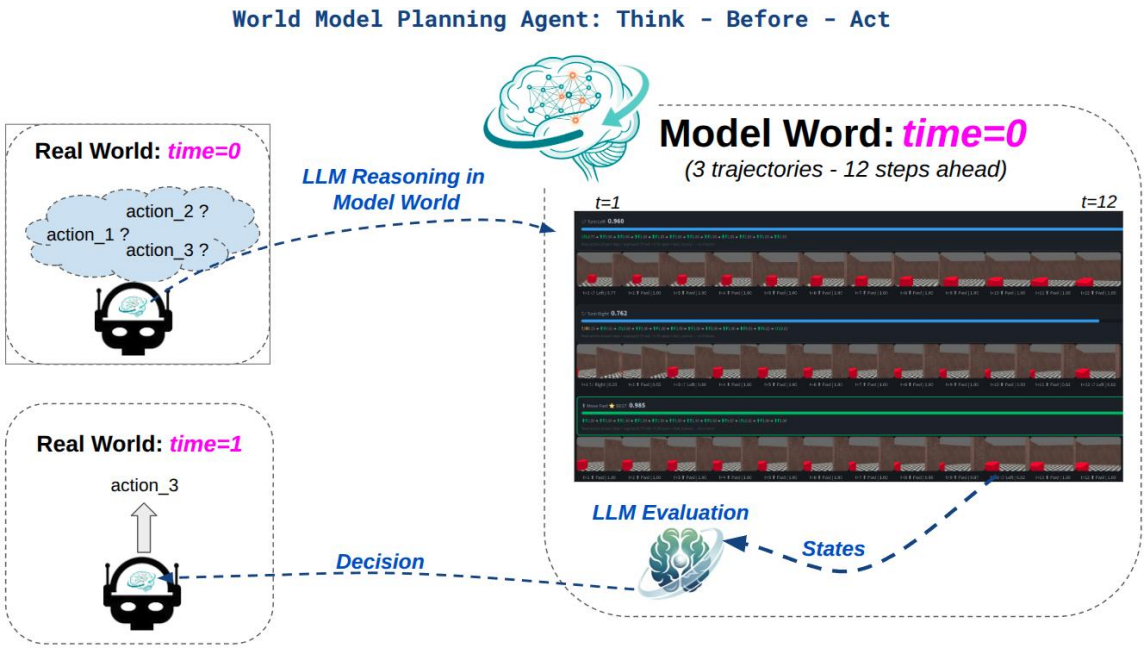

Figure 1: Receding-horizon planning loop: at time t the agent simulates a small set of candidate futures in a *model world*, scores them with an objective function, executes only the first action of the best-scoring plan, and replans at time $t+1$.

In the proposed approach, instead of using the LLM only for next-token prediction, we use it as a high-level planner that queries a model world (simulator or surrogate) to evaluate candidate plans, with MPC selecting the next action.

2 What We Are Trying to Achieve

2.1 MiniWorld Task Goal

The MiniWorld Maze task [1] can be summarised simply: *navigate through a maze to reach a red goal object*. The agent receives an RGB (image) observation I_t from its first-person view and must choose discrete actions (turn left/right, move forward) to reach the goal without getting stuck against walls. Figure 2 shows what the agent sees alongside the bird's-eye view of the maze.

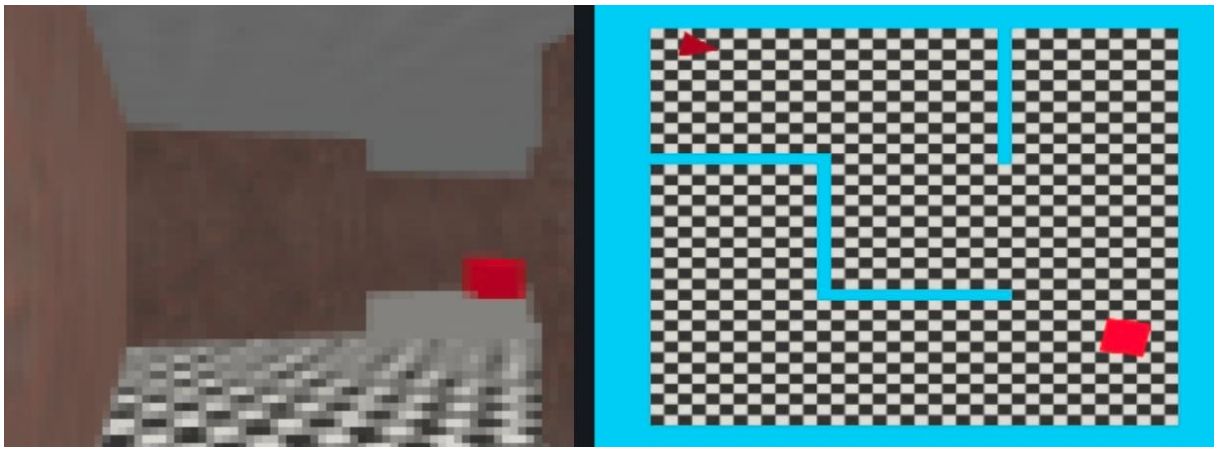

Figure 2: MiniWorld Maze example. Left: first-person RGB observation I_t used by the controller. Right: bird's-eye view of the maze showing the agent location (triangle) and the red goal object (square). The MPC agent seeks actions that increase red-goal visibility and proximity while maintaining open navigable space and avoiding collisions.

2.2 Why MPC?

Single-step greedy decisions from the current image can be deeply misleading. A corridor might look bright and open but end in a wall; the red target might be visible but separated by an obstacle. MPC addresses this by *testing the consequences* of actions over a short simulated future and selecting the action that maximises near-term progress rather than merely looking good in the current frame.

2.3 What the LLM Does (and What It Does Not Do)

In the MiniWorld implementation, dynamics rollouts are produced by the simulator itself, so the core planner can run without an LLM. However, the same "think-before-act" loop naturally supports an LLM-augmented planner:

- **LLM proposes** a small set of candidate plans (e.g., "take the left corridor, avoid dead-ends, keep turning costs low").
- **World model (simulator or surrogate) predicts** what happens if each plan is executed.
- **MPC scores and selects** the best next action (receding horizon).
- **LLM explains** the chosen action in human language (optional, but useful for reporting and after-the-fact review).

This division of labour is especially attractive whenever we want semantic, human-readable reasoning at the top of the stack but still need grounded, repeatable control decisions underneath.

3 System Definition

3.1 State Representation

At time t, the state is defined as:

$$\mathbf{x}_t \;=\; \big(\mathbf{p}_t,\; \theta_t,\; I_t\big) \tag{2}$$

where:

- Agent position (2D): $\mathbf{p}_t = [x_t, y_t]^T$
- Agent heading: $\theta_t \in [0, 2\pi)$
- Observation image: $I_t \in \mathbb{R}^{H\times W\times 3}$ (RGB)

3.2 Action Space

The action space is discrete:

$$\mathcal{A} = \{\text{TURN\_LEFT},\; \text{TURN\_RIGHT},\; \text{MOVE\_FORWARD}\} \equiv \{0,\, 1,\, 2\}. \tag{3}$$

We denote the control input by $\mathbf{u}_t \in \mathcal{A}$.

3.3 Dynamics: True Model f and Surrogate Model $\hat{f}$

The (true) transition function is:

$$\mathbf{x}_{t+1} = f(\mathbf{x}_t, \mathbf{u}_t), \tag{4}$$

where f is the MiniWorld simulator.

In many practical settings the high-fidelity simulator may be expensive, proprietary, or slow. In that case we use a *surrogate dynamics model* $\hat{f}$:

$$\mathbf{x}_{t+1} \approx \hat{f}(\mathbf{x}_t, \mathbf{u}_t). \tag{5}$$

The surrogate $\hat{f}$ can be a reduced-order physics model, a learned predictor trained on simulator traces, or an emulator designed for speed. MPC remains useful because it replans frequently; short-horizon errors in $\hat{f}$ can be corrected by feedback at the next timestep.

3.4 MiniWorld Motion Intuition

For MiniWorld we can interpret the transition as follows:

- Turn_Left / Turn_Right: $\theta_{t+1} = \theta_t \pm \Delta\theta$, position unchanged.
- Move_Forward:

$$\mathbf{p}_{t+1} = \begin{cases} \mathbf{p}_t + \Delta p \cdot [\cos(\theta_t), \sin(\theta_t)]^T & \text{if not blocked} \\ \mathbf{p}_t & \text{if blocked (collision)} \end{cases} \tag{6}$$

4 Image-Based Features (Observation Function)

From the RGB image I_t, the controller extracts two scalar signals:

$$\phi(\mathbf{x}_t) = \begin{bmatrix} r(\mathbf{x}_t) \\ o(\mathbf{x}_t) \end{bmatrix}, \tag{7}$$

where $r(\mathbf{x}_t)$ measures red-object visibility/proximity and $o(\mathbf{x}_t)$ measures path openness (brightness).

4.1 Red Proximity Feature

Define a red-pixel indicator for pixel (i, j):

$$\mathbb{1}_{\text{red}}(i,j) = \begin{cases} 1 & \text{if } I_t(i,j,R) > 150 \ \wedge \ I_t(i,j,G) < 80 \ \wedge \ I_t(i,j,B) < 80 \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

Then the red proximity score is:

$$r(\mathbf{x}_t) = \text{clip}\left(\frac{20}{|I_t|} \sum_{(i,j)\in I_t} \mathbb{1}_{\text{red}}(i,j),\ 0,\ 1 \right), \tag{9}$$

where $|I_t| = H \cdot W$ is the total number of pixels and $\text{clip}(z, 0, 1) = \min(1, \max(0, z))$.

4.2 Path Openness Feature

Define pixel brightness as:

$$\text{brightness}(i,j) = \tfrac{1}{3}\left(I_t(i,j,R) + I_t(i,j,G) + I_t(i,j,B)\right). \tag{10}$$

Then the openness feature is:

$$o(\mathbf{x}_t) = \frac{1}{|I_t|} \sum_{(i,j)\in I_t} \mathbb{1}_{[\text{brightness}(i,j)>80]}. \tag{11}$$

5 Objective Function

5.1 Step Reward / Score

At each simulated step k in a rollout, the controller assigns a step score:

$$s_k = g(\mathbf{x}_k, \mathbf{u}_k) = w_r\, r(\mathbf{x}_k) + w_o\, o(\mathbf{x}_k) + b(\mathbf{x}_k, \mathbf{u}_k). \tag{12}$$

The forward bonus/penalty term is:

$$b(\mathbf{x}_k, \mathbf{u}_k) = \begin{cases} +0.40 & \text{if } \mathbf{u}_k = \text{MOVE\_FORWARD and moved successfully} \\ -0.50 & \text{if } \mathbf{u}_k = \text{MOVE\_FORWARD and blocked (collision)} \\ 0 & \text{if } \mathbf{u}_k \in \{\text{TURN\_LEFT}, \text{TURN\_RIGHT}\}. \end{cases} \tag{13}$$

Weights:

$$w_r = 0.70, \qquad w_o = 0.30. \tag{14}$$

5.2 Finite-Horizon Discounted Objective

For a horizon length N and discount factor $\gamma \in (0, 1)$, the *normalised* discounted objective is:

$$J(\mathbf{x}_t, \mathbf{u}_{0:N-1}) = \frac{\sum_{k=0}^{N-1} \gamma^k \, g(\mathbf{x}_k, \mathbf{u}_k)}{\sum_{k=0}^{N-1} \gamma^k}, \tag{15}$$

with $\mathbf{x}_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k)$ (or $\hat{f}$ when using a surrogate model) and $\mathbf{x}_0 = \mathbf{x}_t$. The normalisation keeps $J \in [0, 1]$ and simplifies the early-termination condition below. The surrogate planning score $S(a_0)$ in Section 11 uses the *unnormalised* form (no denominator) since the argmax is invariant to the positive constant $\sum_k \gamma^k$.

Early termination (goal reached). If the red goal is reached at any step $m < N$ during simulation:

$$J(\mathbf{x}_t, \mathbf{u}_{0:N-1}) = 1.0. \tag{16}$$

6 MPC Optimisation and Receding-Horizon Policy

6.1 MPC Problem (Maximisation Form)

At each real time t, MPC solves:

Receding-Horizon MPC Optimisation

$$\mathbf{u}^*_{0:N-1} = \arg\max_{\mathbf{u}_{0:N-1}} \; J(\mathbf{x}_t, \mathbf{u}_{0:N-1}) \tag{17}$$

$$\text{s.t.} \quad \mathbf{x}_0 = \mathbf{x}_t, \tag{18}$$

$$\mathbf{x}_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k) \;\; (\text{or } \hat{f}), \quad k = 0, \ldots, N-1, \tag{19}$$

$$\mathbf{u}_k \in \mathcal{A}, \quad k = 0, \ldots, N-1. \tag{20}$$

6.2 Receding-Horizon Control Law

Only the first control is executed:

$$\mathbf{u}^*_t = \pi_{\text{MPC}}(\mathbf{x}_t) = \mathbf{u}^*_0, \tag{21}$$

then the system observes $\mathbf{x}_{t+1}$ and replans. This is the feedback loop shown in Figure 1.

6.3 Why MPC Here? (Plain Language + Surrogate Model)

In a maze, locally "good-looking" actions can be misleading: a corridor might be bright but end in a wall, or the red target might be visible but separated by an obstacle. MPC addresses this by testing actions in a short simulated future, letting the agent pick moves that actually lead to progress, not just moves that look good in the current frame.

This look-ahead requires a predictive dynamics model. In MiniWorld, that model is the simulator itself, i.e., $\mathbf{x}_{t+1} = f(\mathbf{x}_t, \mathbf{u}_t)$, so rollouts are accurate [1]. In many real systems the true dynamics may be unavailable or too slow to query. In that case, MPC can be driven

by a surrogate model $\hat{f}$. Because MPC replans at every timestep using fresh observations, it can remain robust even when $\hat{f}$ is imperfect.

6.4 What the Controller Does (Where the Optimisation Happens)

The *controller* is the runtime decision module that maps the current state $\mathbf{x}_t$ to the next executed action $\mathbf{u}_t$. In this project, the controller is an MPC-style planner: it optimises *in the model world* by simulating short-horizon futures and selecting the action that maximises the objective.

Concretely, at each real timestep t the controller evaluates a small set of candidate first actions $\mathbf{u}_0 \in \mathcal{A} = \{\text{TURN\_LEFT}, \text{TURN\_RIGHT}, \text{MOVE\_FORWARD}\}$. For each candidate $\mathbf{u}_0$, it rolls out a length-N trajectory using the dynamics model (the simulator f, or a surrogate $\hat{f}$):

$$\mathbf{x}_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k) \quad (\text{or } \hat{f}), \qquad \mathbf{x}_0 = \mathbf{x}_t. \tag{22}$$

Each rollout is scored with the discounted objective $J(\mathbf{x}_t, \mathbf{u}_{0:N-1})$. The optimisation step is the argmax over candidates:

$$\mathbf{u}_t^* = \arg\max_{\mathbf{u}_0 \in \mathcal{A}} J(\mathbf{x}_t, \mathbf{u}_{0:N-1}(\mathbf{u}_0)) \,. \tag{23}$$

Finally, only the first action $\mathbf{u}_t^*$ is executed in the real environment; at time $t{+}1$ the controller replans from the new observation.

Figure 3: Example rollout evaluation at a single real timestep. Three candidate trajectories are simulated in the model world, corresponding to the three possible first actions (Turn Left, Turn Right, Move Forward). Each rollout is scored by the discounted objective J. The controller selects the first action of the highest-scoring rollout (highlighted as BEST) and replans at the next real timestep.

7 Approximate MPC Implementation: 3 Trajectories

Exhaustive search over all action sequences would require evaluating $|\mathcal{A}|^N = 3^N$ trajectories. Instead, the implementation evaluates exactly three candidate rollouts, one for each possible

first action:

$$\mathbf{u}_0 \in \{\text{TURN\_LEFT},\ \text{TURN\_RIGHT},\ \text{MOVE\_FORWARD}\}. \tag{24}$$

For each candidate first action, the remaining steps are chosen greedily (optionally using a U-turn prevention rule), producing a rollout $\mathbf{u}_{0:N-1}(\mathbf{u}_0)$. The chosen action is then:

$$\mathbf{u}_t^* \;=\; \arg\max_{\mathbf{u}_0 \in \mathcal{A}} \; J(\mathbf{x}_t, \mathbf{u}_{0:N-1}(\mathbf{u}_0))\,. \tag{25}$$

This yields MPC-style receding-horizon control with an *approximate optimizer* (greedy rollouts), which is often a good trade-off when the action space is small and the simulator is fast.

Greedy MPC

At every horizon step, all 3 actions are evaluated via the score head and the best is selected. The surrogate's own predictions determine every action at every step - this is what MPC actually means.

8 System Architecture

8.1 WorldModelNet Architecture

`WorldModelNet` is a neural network that learns to simulate the world - it takes a camera image and an action as input, and **predicts what the world will look like next**.

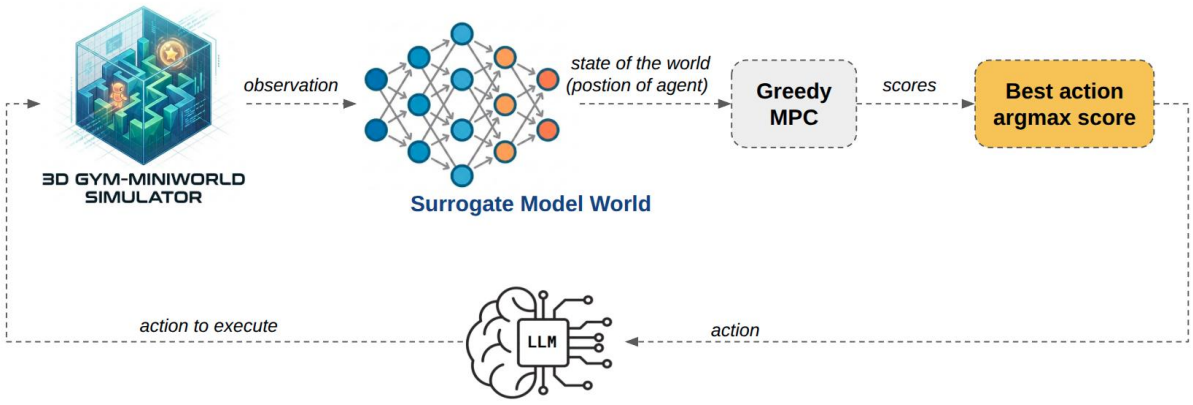

Figure 4: The MiniWorld surrogate pipeline. At every decision step the current observation is encoded into a compact latent vector by the CNN encoder. The greedy MPC kernel then evaluates all three possible actions over a seven-step horizon, selecting the action whose predicted score is highest at each step. The entire planning process runs inside the neural network and takes (in our case) 3.8 ms - the real environment is only called once, to execute the chosen action.

Step 1 - Encoder. The network first looks at the current image (60×80 pixels, colour). Three convolutional layers compress it from raw pixels into a compact 256-dimensional summary vector $\mathbf{z}_t$, called the *latent vector*. Think of it as the network forming a mental picture of the current situation.

Step 2 - Transition. The network then asks: *if I take this action, what happens next?* It takes the latent vector $\mathbf{z}_t$ together with the chosen action and predicts the next latent

vector $\mathbf{z}_{t+1}$ - without ever rendering a new image. This is the *dynamics model*, the core of the world model.

Step 3 - Decoder. If needed, the next latent vector can be decoded back into a full predicted image - the network reconstructs what the camera would show after the action has been taken.

Step 4 - Prediction Heads. On top of $\mathbf{z}_{t+1}$, three small sub-networks make fast predictions: did the agent reach the goal? did it actually move forward? and what is the expected reward score? These heads allow the planner to evaluate actions without decoding images at all.

The key insight is that once trained, the entire pipeline runs in a few milliseconds. The agent can mentally simulate hundreds of future steps and select the best action before touching the real environment.

The network f_θ consists of four sub-networks:

$$\mathbf{z}_t = \mathrm{Enc}_\phi(o_t) \in \mathbb{R}^{256} \tag{26}$$
$$\mathbf{z}_{t+1} = \mathrm{Trans}_\psi([\mathbf{z}_t;\, \mathbf{e}_a]) \in \mathbb{R}^{256} \tag{27}$$
$$\hat{o}_{t+1} = \mathrm{Dec}_\xi(\mathbf{z}_{t+1}) \in [0,1]^{3\times 60\times 80} \tag{28}$$
$$(\hat{r}_{\mathrm{red}},\, \hat{r}_{\mathrm{open}}) = \mathrm{Score}_\omega(\mathbf{z}_{t+1}) \in [0,1]^2 \tag{29}$$

where $\mathbf{e}_a = \mathrm{Emb}(a) \in \mathbb{R}^{16}$ is a learned action embedding ($|\mathcal{A}| = 3$).

Encoder $\mathbf{Enc}_\phi$. Four stride-2 convolutions with batch normalisation and ReLU, followed by global average pooling and a linear projection:

$$[0,1]^{3\times 60\times 80} \xrightarrow{\text{Conv(32)}\to\text{Conv(64)}\to\text{Conv(128)}\to\text{Conv(128)}} \mathbb{R}^{256}$$

Transition model $\mathbf{Trans}_\psi$. A four-layer residual Multilayer Perceptron (MLP) with LayerNorm:

$$\mathbb{R}^{272}\ (=256{+}16) \xrightarrow{\text{Linear}\times 4,\ \text{LayerNorm, ReLU}} \mathbb{R}^{256}$$

Decoder $\mathbf{Dec}_\xi$. Transposed convolutions with bilinear upsampling (used for display only, never during the planning loop):

$$\mathbb{R}^{256} \xrightarrow{\text{Linear, ConvT}\times 4,\ \text{upsample}} [0,1]^{3\times 60\times 80}$$

Score head $\mathbf{Score}_\omega$. Shallow MLP with sigmoid output (used for all MPC scoring):

$$\mathbb{R}^{256} \xrightarrow{\text{Linear}(256\to 128),\ \text{ReLU},\ \text{Linear}(128\to 2),\ \sigma} [0,1]^2$$

Auxiliary heads. Two binary heads attached to $\mathbf{z}_{t+1}$:

$$\hat{p}_{\mathrm{term}} = \sigma(\mathrm{MLP}_{16}(\mathbf{z}_{t+1})) \qquad \text{(goal reached?)}$$
$$\hat{p}_{\mathrm{moved}} = \sigma(\mathrm{MLP}_{16}(\mathbf{z}_{t+1})) \qquad \text{(not blocked by wall?)}$$

8.2 Architecture Diagram

Figure 5 shows the full `WorldModelNet` data flow with all tensor dimensions for the MiniWorld implementation.

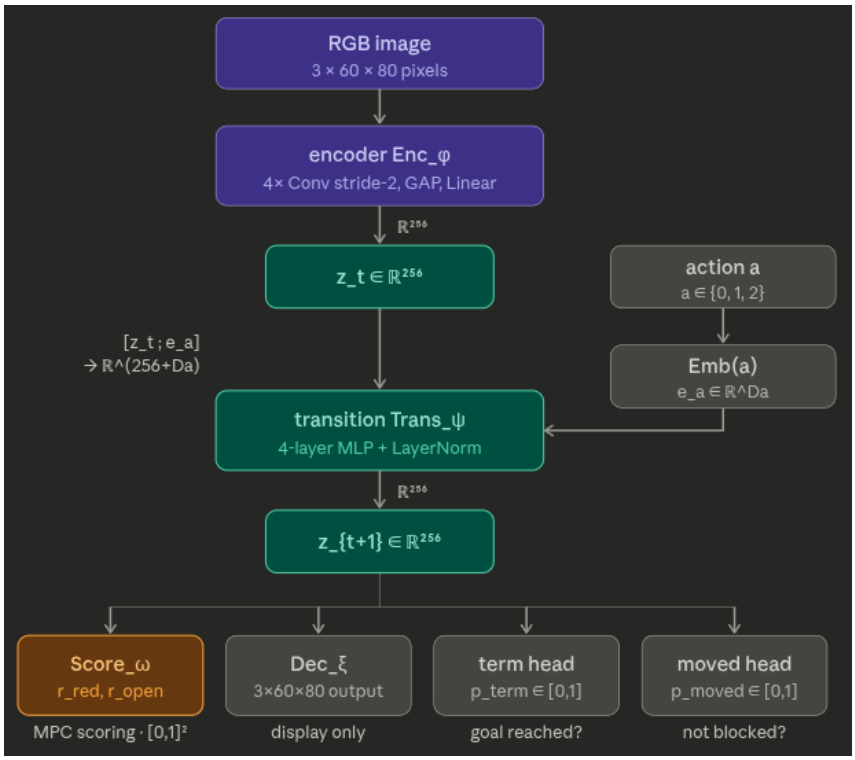

Figure 5: `WorldModelNet` surrogate architecture (MiniWorld). **Purple path:** the CNN encoder compresses a 3×60×80 RGB image into a 256-dimensional latent vector $\mathbf{z}_t$. **Teal path:** the transition model Trans_ψ takes the concatenation $[\mathbf{z}_t; \mathbf{e}_a] \in \mathbb{R}^{256+D_a}$ and predicts the next latent $\mathbf{z}_{t+1} \in \mathbb{R}^{256}$, keeping the entire planning loop inside the latent space. **Amber head** (Score_ω): the only head called during MPC planning; outputs $(r_{\text{red}}, r_{\text{open}}) \in [0,1]^2$. **Gray heads**: the decoder Dec_ξ is used for display only; the two binary heads (goal reached, not blocked) are auxiliary training signals.

8.3 Pipeline Overview

With the surrogate trained, the agent replaces the real simulator entirely during planning. At every decision step the pipeline runs four stages:

1. **Encode once:** the current observation o_t is passed through Enc_ϕ to produce the latent vector $\mathbf{z}_t \in \mathbb{R}^{256}$- a single forward pass, shared across all candidate actions.
2. **Fused planning:** `plan_greedy_mpc` evaluates all A=3 first actions over a horizon H=7, greedily expanding the best continuation at each step via the score head Score_ω - no pixel decode, no pre-written plans.
3. **Select & reason:** the best action a^* and its cumulative score S_{a^*} are returned; **the LLM uses these for natural-language justification if required**.
4. **Commit:** only a^* is executed in the real environment; o_{t+1} is returned and the cycle repeats.

9 Data Collection

9.1 Collection Policy

Transitions are collected using a mixed behavioural policy:

$$\pi_{\text{collect}}(a \mid o) = \begin{cases} \pi_{\text{greedy}}(a \mid o) & \text{with probability } 0.70 \\ \mathcal{U}(\mathcal{A}) & \text{with probability } 0.30 \end{cases} \tag{30}$$

The 30% random component ensures coverage of wall-collision states and dead-ends that a purely greedy policy would rarely visit. This diversity is critical: in our 81,086-transition dataset only 19.1% of steps physically moved the agent forward, so random actions are essential to balance the dataset.

9.2 Stored Transition Format

Each transition τ_t stores:

$$\tau_t = \big(o_t,\ a_t,\ o_{t+1},\ \text{term}_t,\ \text{moved}_t,\ r_t^{\text{red}},\ r_t^{\text{open}}\big)$$

Field	Type / Shape	Description
o_t	`uint8` $60\times80\times3$	RGB observation
a_t	`int` $\in \{0,1,2\}$	Turn-left / Turn-right / Fwd
o_{t+1}	`uint8` $60\times80\times3$	Next RGB observation
term_t	`bool`	Goal reached
moved_t	`bool`	Agent not wall-blocked
r_t^{red}	`float32` $\in [0,1]$	Fraction of red pixels in o_{t+1}
r_t^{open}	`float32` $\in [0,1]$	Fraction of non-wall pixels in o_{t+1}

10 Training

10.1 Loss Function

Training minimises a four-term weighted loss:

$$\mathcal{L} = \underbrace{\lambda_{\text{img}}\,\mathcal{L}_{\text{img}}}_{\text{pixel recon.}} + \underbrace{\lambda_{\text{term}}\,\mathcal{L}_{\text{term}}}_{\text{goal detect.}} + \underbrace{\lambda_{\text{moved}}\,\mathcal{L}_{\text{moved}}}_{\text{wall detect.}} + \underbrace{\lambda_{\text{score}}\,\mathcal{L}_{\text{score}}}_{\text{MPC reward}} \tag{31}$$

with weights $(\lambda_{\text{img}}, \lambda_{\text{term}}, \lambda_{\text{moved}}, \lambda_{\text{score}}) = (1,\ 5,\ 2,\ 3)$.

$$\mathcal{L}_{\text{img}} = \frac{1}{B}\sum_i \|\hat{o}_{t+1}^{(i)} - o_{t+1}^{(i)}\|_2^2 \tag{32}$$

$$\mathcal{L}_{\text{term}} = -\frac{1}{B}\sum_i \big[\text{term}^{(i)} \log \hat{p}_{\text{term}}^{(i)} + (1-\text{term}^{(i)}) \log(1-\hat{p}_{\text{term}}^{(i)})\big] \tag{33}$$

$$\mathcal{L}_{\text{moved}} = -\frac{1}{B}\sum_i \big[\text{moved}^{(i)} \log \hat{p}_{\text{moved}}^{(i)} + (1-\text{moved}^{(i)}) \log(1-\hat{p}_{\text{moved}}^{(i)})\big] \tag{34}$$

$$\mathcal{L}_{\text{score}} = \frac{1}{B}\sum_i \big\|[\hat{r}_i^{\text{red}}, \hat{r}_i^{\text{open}}] - [r_i^{\text{red}}, r_i^{\text{open}}]\big\|_2^2 \tag{35}$$

The high weight on $\mathcal{L}_{\text{term}}$ ($\lambda = 5$) reflects the critical importance of goal detection: a missed goal-state prediction would cause the agent to discard a winning action.

10.2 Training Procedure

Hyperparameter	Value
Optimiser	Adam, $\beta = (0.9, 0.999)$, weight decay 0
Learning rate	3×10^{-4}, halved (ReduceLROnPlateau, patience=5)
Batch size	256 (stable gradients; smaller sizes cause noisy val loss)
Train/val split	90% / 10%, fixed random seed
Gradient clip	$\ell_\infty = 1.0$
Early stopping	15 epochs without val improvement
Latent dim	256
Action emb.	16-dim learned embedding
Best epoch	6 (val_loss = 0.0694)
Dataset size	200 episodes

10.3 Training Curves

Figure 6 shows the validation loss history exported from TensorBoard. The best checkpoint is automatically saved at epoch 6 (val_loss = 0.0694) and used for all inference.

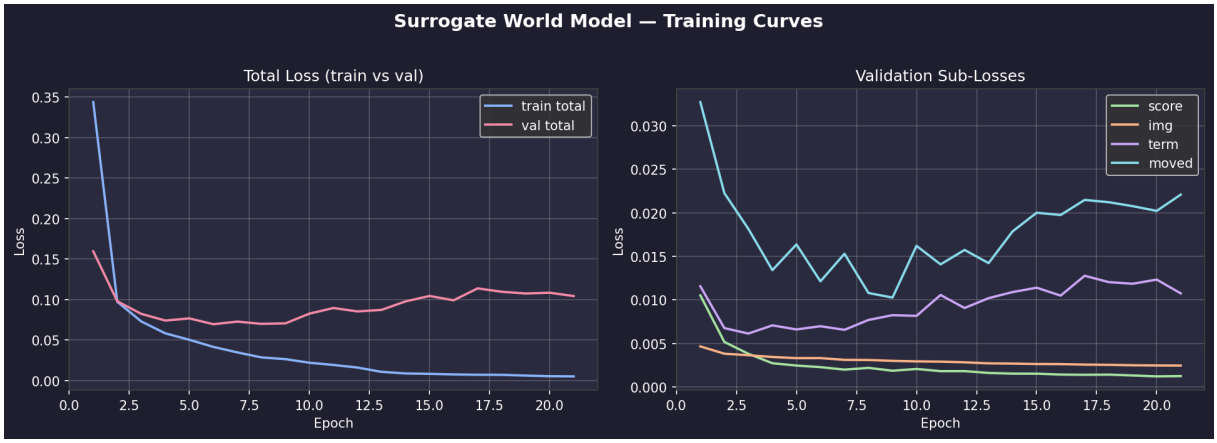

Figure 6: **Left:** Total training and validation loss. The orange dashed line marks the best epoch (6); the model is saved here and training continues only to confirm no further improvement. The rising validation loss after epoch 8 is classical overfitting on an 81k-sample dataset - the saved checkpoint is unaffected. **Right:** Validation sub-losses. The image MSE (blue) converges steadily; score regression (orange) reaches near-zero quickly and stays stable; binary heads (red/purple) converge within 5 epochs.

11 Inference and MPC Planning

11.1 MPC Objective and Score Function

Once the surrogate model is trained, it replaces the real simulator entirely during planning. The agent runs all candidate actions through the neural network, compares the predicted outcomes, and only then commits to the best choice in the real world. This separation between *thinking* (planning in the surrogate) and *acting* (executing in the real environment) is the central idea of the pipeline.

The score head Score_ω maps a latent state directly to the two reward components defined in Section 4. The composite reward used throughout the planning loop is:

$$\hat{R}(\mathbf{z}) = w_r\, \hat{r}_{\text{red}}(\mathbf{z}) + w_o\, \hat{r}_{\text{open}}(\mathbf{z}), \qquad w_r = 0.70, \quad w_o = 0.30 \tag{36}$$

where $(\hat{r}_{\text{red}}, \hat{r}_{\text{open}}) = \text{Score}_\omega(\mathbf{z}) \in [0,1]^2$ (Eq. 29). This is consistent with the step-score weights defined in Section 5.

At decision step t, with $\hat{\mathbf{z}}_0 = \text{Enc}_\phi(o_t)$, the planner selects the first action that maximises the greedy discounted sum over horizon H:

$$a^* = \arg\max_{a_0 \in \mathcal{A}} \; S(a_0) \tag{37}$$

$$S(a_0) = \sum_{h=0}^{H-1} \gamma^h\, \hat{R}\big(\hat{\mathbf{z}}_{h+1}^{a_0}\big)\,, \qquad \gamma = 0.95 \tag{38}$$

where the latent trajectory $\{\hat{\mathbf{z}}_h^{a_0}\}$ is built as follows: the first step uses the chosen a_0; each subsequent step selects the locally best continuation:

$$\hat{\mathbf{z}}_1^{a_0} = \text{Trans}_\psi([\hat{\mathbf{z}}_0;\, \mathbf{e}_{a_0}]) \tag{39}$$

$$\hat{\mathbf{z}}_{h+1}^{a_0} = \text{Trans}_\psi([\hat{\mathbf{z}}_h^{a_0};\, \mathbf{e}_{\bar{a}_h}])\,, \quad h = 1, \ldots, H-1 \tag{40}$$

$$\bar{a}_h = \arg\max_{a \in \mathcal{A}} \; \hat{R}(\text{Trans}_\psi([\hat{\mathbf{z}}_h^{a_0};\, \mathbf{e}_a])) \tag{41}$$

The key advantage is speed: evaluating H steps in the surrogate takes a few milliseconds, whereas running the same rollout in the real simulator would be orders of magnitude slower. No pixel decoding is required at any stage.

11.2 Greedy MPC Kernel: `plan_greedy_mpc`

The key to high speed is evaluating all trajectories in a *single* compiled PyTorch kernel call with no pixel decode. Rather than pre-writing action sequences, `plan_greedy_mpc` computes each action from the surrogate's own score head-this is what MPC actually means.

For each of A=3 initial first-actions, one latent beam is maintained. At each subsequent step $h = 2, \ldots, H$, the beam is expanded with all A=3 possible actions; the best continuation is kept via the greedy argmax (Eq. 41):

$$\bar{a}_h = \arg\max_{a \in \mathcal{A}} \; \hat{R}(\text{Trans}_\psi([\mathbf{z};\, \mathbf{e}_a])) \tag{42}$$

The cumulative discounted score for beam i (first action $a_0{=}i$) is (Eq. 38):

$$S_i = \sum_{h=1}^{H} \gamma^{h-1}\, \hat{R}\Big(\hat{\mathbf{z}}_h^{(i)}\Big) \tag{43}$$

The first action of the highest-scoring beam is executed.

Algorithm 1 PLAN_GREEDY_MPC($\hat{\mathbf{z}}_0,\ H,\ \mathcal{A},\ \gamma$)

Require: latent $\hat{\mathbf{z}}_0 \in \mathbb{R}^{256}$, horizon $H{=}7$, actions $|\mathcal{A}|{=}3$, discount $\gamma{=}0.95$
Ensure: best first action $a^* \in \mathcal{A}$

for $a_0 = 0, \ldots, |\mathcal{A}| - 1$ **do** ▷ one beam per first action
  $\mathbf{z} \leftarrow \mathrm{Trans}_\psi([\hat{\mathbf{z}}_0;\ \mathbf{e}_{a_0}])$ ▷ Eq. 39
  $S_{a_0} \leftarrow \hat{R}(\mathbf{z})$; $\delta \leftarrow \gamma$
  for $h = 2, \ldots, H$ **do**
    $\bar{a} \leftarrow \arg\max_{a \in \mathcal{A}}\ \hat{R}(\mathrm{Trans}_\psi([\mathbf{z};\ \mathbf{e}_a]))$ ▷ Eq. 42
    $\mathbf{z} \leftarrow \mathrm{Trans}_\psi([\mathbf{z};\ \mathbf{e}_{\bar{a}}])$ ▷ Eq. 40
    $S_{a_0} \leftarrow S_{a_0} + \delta\, \hat{R}(\mathbf{z})$; $\delta \leftarrow \delta \cdot \gamma$ ▷ Eq. 43
  end for
end for
return $a^* = \arg\max_{a_0 \in \mathcal{A}}\ S_{a_0}$ ▷ Eq. 37

Compute cost. $|\mathcal{A}| \times |\mathcal{A}| \times H = 3 \times 3 \times 7 = 63$ transition calls total per planning step. No pixel decoding is invoked at any point; Dec_ξ is only used for display thumbnails outside the planning loop.

11.3 Inputs and Outputs at Inference

Component	Input	Output
Encoder	$o_t \in \{0 \ldots 255\}^{60\times80\times3}$	$\mathbf{z}_t \in \mathbb{R}^{256}$
Transition	$(\mathbf{z}_h,\ a) \in \mathbb{R}^{256} \times \mathcal{A}$	$\mathbf{z}_{h+1} \in \mathbb{R}^{256}$
Score head	$\mathbf{z}_{h+1}$	$(r_{\text{red}}, r_{\text{open}}) \in [0,1]^2$
plan_greedy_mpc	$(\mathbf{z}_0, H{=}7, A{=}3)$	$a^* \in \mathcal{A}$
Decision	beam scores S_0, S_1, S_2	$a^* = \mathrm{argmax}(S_i)$
Decoder	$\mathbf{z}_{h+1}$ *(optional)*	$\hat{o} \in [0,1]^{3\times60\times80}$ *(display only)*

12 Experimental Results

12.1 Prediction Accuracy

Benchmarked on 20 held-out episodes of `MiniWorld-MazeS3-v0`:

Metric	Value	Interpretation
Pixel MSE ↓	0.00279	Per-pixel squared error
PSNR ↑	26.51 dB	>25 dB = visually acceptable
Terminated accuracy ↑	100.0%	Goal-reached classification
Moved accuracy ↑	97.7%	Wall-collision classification
Red-score MAE ↓	0.006	Goal visibility regression
Open-score MAE ↓	0.022	Corridor openness regression
MPC score error	< 1.1%	Planning quality
Per-action pixel MSE:		
TURN_LEFT	0.00175	Minimal visual change
TURN_RIGHT	0.00171	Minimal visual change
MOVE_FORWARD	0.00290	Larger visual change (motion)
Validation loss sub-components (best epoch 6):		
val_total	0.0694	Combined weighted loss
val_img	0.00244	Image reconstruction MSE
val_score	0.00127	Score head MSE
val_term	0.00321	Termination BCE
val_moved	0.01165	Moved BCE

12.2 Benchmark Dashboard

Figure 7 shows the full benchmark dashboard with prediction accuracy, per-action breakdown, and speed comparison.

12.3 Planning Speed

Table 1: Planning call latency (100 calls, 40-call warm-up, CPU). Each call covers A=3 first-actions × H=7 steps = 63 surrogate transitions.

Path	ms/call	calls/s	Speedup
Real simulator (21 sequential env steps)	94.4	10.6	0.28×
With pixel decode	13.1	76.3	2.02×
Greedy MPC, no decode (`plan_greedy_mpc`)	**3.8**	**263**	**7.02**×
Real simulator (single step, reference)	26.5	37.7	1.00×

12.4 Latency Decomposition

Component	Time (μs)	Share
CNN encoder (60×80 image)	430	11%
`plan_greedy_mpc` (A=3, H=7)	620	16%
Python wrapper / tensor overhead	2750	72%
Total	**3800**	100%

The 72% Python overhead is a *cost* of the Python runtime and is the primary barrier to higher speedup on fast environments like MiniWorld.

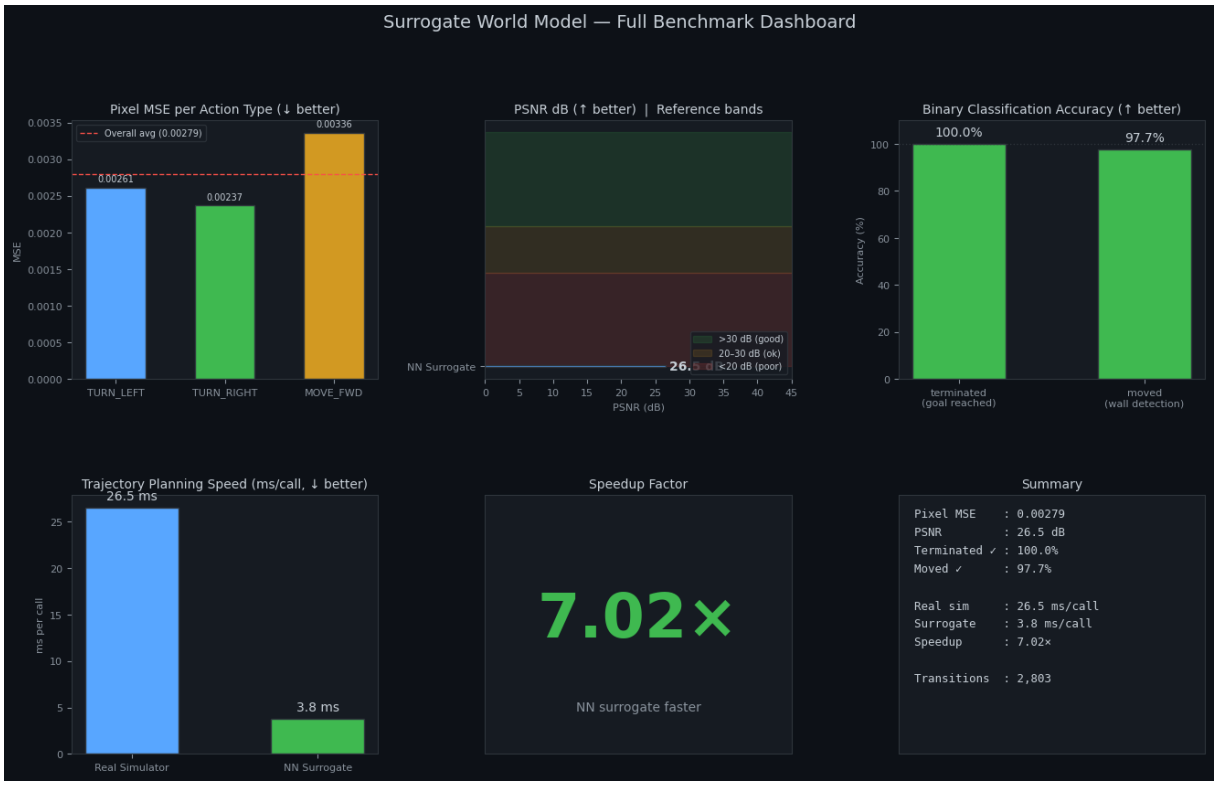

Figure 7: Surrogate world model benchmark dashboard (`MiniWorld-MazeS3-v0`, 20 episodes, 2,414 transitions). *Top row:* pixel MSE, PSNR, and binary head accuracy. *Bottom row:* per-action MSE and planning speed for all three paths.

13 Conclusion

What started as a small experiment - can a maze-navigating agent get away with imagining the outcome of its actions instead of actually trying them? - turned into a fairly complete answer. The receding-horizon MPC loop built here does exactly what the name promises: at every step it looks a handful of moves ahead inside a model of the world, scores what it sees, and only ever commits to the first action of whichever plan looks best. Replanning at every timestep is what makes this forgiving of an imperfect model, which matters because the model here, `WorldModelNet`, is not the real MiniWorld simulator at all but a small convolutional network trained to imitate it.

The surrogate turned out to be a good enough stand-in that the difference is barely visible in the numbers: prediction error stays low across every head we measured, goal detection is essentially perfect, and the planner's own scoring matches what the true simulator would have reported to within about one percent. What changes dramatically is speed. Querying the real simulator 63 times per planning step - three candidate first actions times a seven-step horizon - costs on the order of 94 milliseconds; querying the network instead costs 3.8, a 7× speedup that comes entirely from replacing an expensive renderer with a forward pass through a small MLP. The latency breakdown is honest about where that time still goes: barely a quarter of it is the network itself, and the rest is Python and tensor-handling overhead, which is the obvious place to look if this needs to get faster still.

None of this is complicated by design. The action space is three discrete moves, the reward is two numbers read off a rendered image, and the planner is a greedy beam search rather than a full trajectory optimiser. That simplicity is the point: it is much easier to trust a

result on a system small enough to fully understand, and the pieces that make it work - a learned latent transition model, a score head that never needs to decode a pixel to make a decision, and a receding-horizon controller that never over-commits to its own predictions - do not depend on the maze being small. They are the same pieces one would reach for in a larger, messier environment where running the real simulator is the actual bottleneck. This note is the record of getting those pieces working cleanly on a problem small enough to check every number by hand, and that was always the point of doing it on MiniWorld first.

References

[1] Miniworld: Maze environment. https://miniworld.farama.org/environments/maze/.

[2] World models. https://worldmodels.github.io/.

[3] Emmanuel Dupoux, Yann LeCun, and Jitendra Malik. Why ai systems don't learn and what to do about it: Lessons on autonomous learning from cognitive science. 2026. https://arxiv.org/pdf/2603.15381.

[4] Grady Williams, Andrew Aldrich, and Evangelos A. Theodorou. Model predictive path integral control: From theory to parallel computation. *Journal of Guidance, Control, and Dynamics*, 40(2):344–357, 2017.

[5] Quentin Garrido, Tushar Nagarajan, Basile Terver, Nicolas Ballas, Yann LeCun, and Michael Rabbat. Learning latent action world models in the wild. 2026. https://arxiv.org/pdf/2601.05230.