Proceedings of Machine Learning Research **318** (2026)
The 39th Canadian Conference on Artificial Intelligence

CA2: Code-Aware Agent for Automated Game Testing

Valliappan Chidambaram Adaikkappan[†◇], Vincent Martineau [‡], Joshua Romoff [‡], David Meger[†◇]
[†] McGill University
[‡] Ubisoft
[◇] Mila Quebec AI Institute

Abstract

Automated game testing is important for verifying game functionality, but it remains a costly and time-consuming process. Manual testing often misses edge cases, and current automated methods struggle to provide full code coverage. Prior work has explored reinforcement learning (RL) for game testing, but without leveraging internal code signals such as the call stack. We present Code-Aware Agent (CA2), which uses call stack information to learn effective testing strategies. The agent receives the current function call trace along with the game state and learns to reach specific target functions. We instrument two types of environments, 1) State-based and 2) Image-based, with support for efficient call stack extraction. Through experimental evaluation, we find that CA2 achieves consistent improvement over the non-code aware baselines, which does not leverage call stack information. Our results show that incorporating code signals like the call stack enables more effective and targeted game testing.

Keywords: Reinforcement Learning, Representation Learning, Game-Testing.

1. INTRODUCTION

Automated game testing is a crucial part of game development, helping developers ensure quality, stability, and performance. However, despite its importance, it remains one of the most tedious and labor-intensive tasks, often requiring large teams of human testers [1]. Recent trends show that the use of AI tools to aid traditional testing practices, such as manual testing, scripted coverage procedures, and basic unit tests [2]. RL agents are potentially excellent candidates as they have been shown to simulate complex human behavior in different video game settings [3, 4]. However, the effectiveness of an automated testing approach that simply replicates human behavior remains largely unexplored. Most prior work in automated game testing focuses on maximizing *state coverage*, ensuring agents visit a wide range of game states [5–7]. However, identical game states can trigger diverse function calls based on the chosen action. As a result, state coverage alone fails to guarantee meaningful code coverage.

This paper proposes new research tools and methods relevant for *function-level RL-based game testing*, where agents are trained for a functional test that aims to improve code coverage. Specifically, we present two novel environments designed for functional testing of games, which adhere to the gym environments framework [8]. The first environment builds upon a multilevel Godot game that has been explored in previous RL research [9]. The second transforms the very well-known Crafter [10] RL benchmark into a game-testing scenario. Unlike traditional RL environments, we define sparse rewards for reaching functions, which requires the integration of a source code profiler into the traditional gym interface.

Running real video game engines with a profiler can be slow and difficult to scale for online RL training that requires millions of samples [3]. Thus, we have collected a corpus of expert data gathered by human testers, allowing the use of imitation learning (IL) or offline RL

Correspondence to: valliappan.chidambaramadaikkappa@mail.mcgill.ca

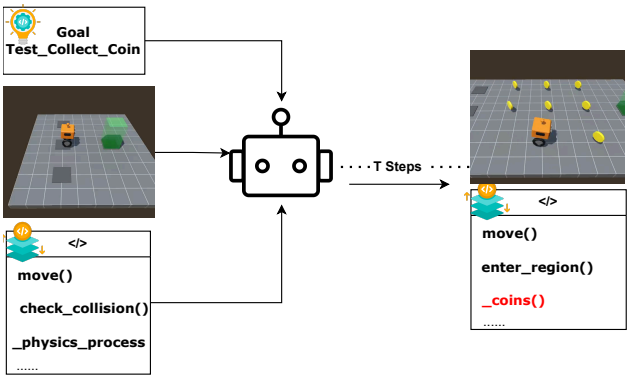

Figure 1. Presenting the Code-Aware Agent (CA2) for automated game testing. It seamlessly tests low-level game functions by leveraging call stack information.

algorithms that do not need to directly interact with the game engine for training.The integration of a source code profiler into our environments allows RL agents to be **code-aware** while interacting with the game. In addition to typical observations that include images or other transcriptions of the game's simulated physical space, we have experimented with agents that can make use of the call stack as an additional observation.

To this end, we propose *Code-Aware Agent* , which can understand and test low-level game-state functions by leveraging call stack information. In our approach, agents play the game and try to reach a targeted method in the source code, thus exploring execution branches, exercising the code base, discovering bugs or crashes, and verifying performance characteristics. We hypothesize that by incorporating programmatic feedback, such as the call stack, RL can outperform traditional code-unaware agents in terms of reaching specific function-level goals. This makes the function-coverage goals observable to the agent, but raises the challenge of multi-modality. The function stack and physical observations have significantly different dimensionalities and unique transition dynamics. We find that naive integration of these state elements does not lead to strong agent performance, but through careful joint representations, our code-aware agents give a boost to testing efficacy. Our approach proves effective across both pixel-based environments with discrete action spaces and state-based environments with continuous actions.

Our main contributions are as follows:

- We introduce a suite of two function-level game testing environments, Crafter and Godot, each instrumented with a lightweight source code profiler and equipped with expert gameplay trajectories to support offline reinforcement learning.
- A novel code-aware agent design that augments standard offline IL/RL architectures with call stack observations. See Figure 1.
- The proposed approach is engine-agnostic by design. We demonstrate this through two distinct implementations: Crafter, a pixel-based environment built with the Pygame engine (Python). Multilevel Robot Game, implemented in Godot (GDScript).

2. RELATED WORK

Automated Game testing. Several studies have investigated the use of AI in game testing. [11] propose an object detection-based approach for detecting non-crashing game glitches, such as visual glitches across 20 real-world games. This approach relied on a data

augmentation approach that can generate game images from User Interface (UI) glitches. Recent advances in large language models (LLMs) have spurred the development of generalist agents for complex video game environments such as StarCraft II[12], Minecraft [13], and Civilization [14]. Early systems often rely on structured APIs or predefined action spaces to interface with games, which simplifies control but limits generalization across environments. End-to-end approaches have demonstrated greater flexibility: VPT[15] showed that agents can learn directly from raw video via large-scale pretraining on human gameplay, though such datasets are expensive to collect and difficult to scale. Similarly, SIMA [16]trained embodied agents across multiple 3D games using behavior cloning, but remained constrained by high data requirements and limited transfer. More recent work emphasizes autonomous skill acquisition and continual learning. VOYAGER [17] introduced self-generated curricula to enable open-ended exploration in Minecraft, while Cradle leveraged multimodal LLMs to operate in visually rich, unstructured games such as Red Dead Redemption 2. Beyond game-specific agents, several efforts have explored generalist interaction with arbitrary user interfaces and environments, as well as the use of LLM agents to model social behaviors and multi-agent coordination in virtual worlds.[18–20]

Reinforcement learning Assisted Gameplay Testing. Several works proposed the use of Reinforcement Learning for automated game testing. [21], presented *Wuji*, which aims to generate effective policies that explore more game states where bugs may be present, using evolutionary multi-objective optimization (EMOO) and deep reinforcement learning (DRL). Similarly, Agarwal, Herrmann, Wallner, and Beck [22] used RL and an evolutionary algorithm to understand the navigation behavior in level design, with the main contribution being the web-based visualization tool. This work was focused on a specific 2D game called *Sonic the Hedgehog 2*. Recent works have used exploration rewards for game testing, [23] designed an environment with bugs. RL agent with reward signals to navigate to explore new states, showing an increase in test coverage. Similar to this work, [5–7, 24] used curiosity-based intrinsic reward for exploration, prioritizing state coverage for automated game-testing. Especially, [6] proposed an approach to train agents capable of exploring while imitating an expert demonstration. EVOLUTE [25] is a recent approach that uses imitation learning by combining behavioral cloning with energy-based models to handle both discrete and continuous actions.

Previous work in automated game testing has largely focused on maximizing *state coverage* [5–7]. Although this strategy can surface untested areas of the environment, it does not guarantee adequate coverage of the underlying codebase. To our knowledge, we are the first to incorporate call stack information into reinforcement learning for automated game testing.

3. PRELIMINARIES

Problem Statement: This is an offline learning approach, and the agent's goal is to learn and successfully reach the functions of interest from the existing data without the privilege of learning from online exploration. For this work, we only have access to a fixed, limited dataset consisting of a trajectory rolled out by game testers.

Offline Reinforcement Learning: A code-aware MDP is described by a tuple (S, C, A, P, G) where (S, A, P) are the state space, action space, and transition of the agent's MDP respectively, C is a finite set of code functions, and $G \subset C$ is a set of goal (target) functions. The code-aware MDP induces a new MDP $(S \times S^{\mathsf{code}}, A, P)$ where S^{code} is a space of call stacks. The agent in this MDP will be goal-conditioned, given a goal $g \in G$, it tries to reach a

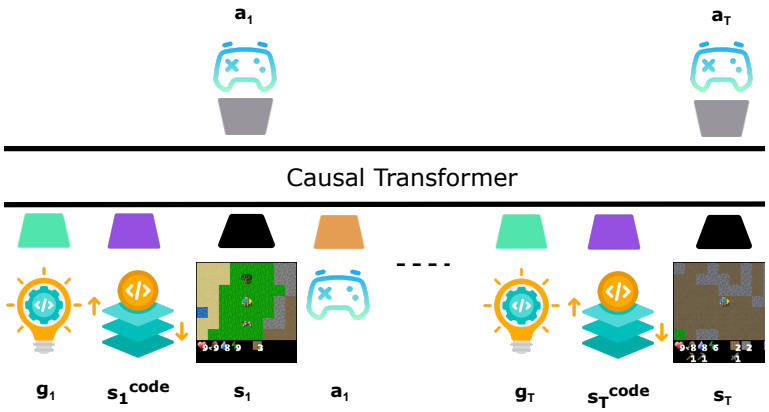

Figure 2. Illustration of our Code-Aware Agent. All architectures use similar input formats, where a goal g, call stack s_t^{code}, environment state s_t, and action a_t are embedded and used as input.

state $x = (s, s^{\text{code}})$ such that $g \in s^{\text{code}}$. The length of the number of function calls can be of arbitrary length, varying from one timestep to another based on the function calls. The trajectory τ is represented as follows $(s_0, s_0^{\text{code}}, a_0, s_1, s_1^{\text{code}}, a_1 \cdots s_T, s_T^{\text{code}}, a_T)$, where s_t^{code} is a list of function call represented as $[f_t^1, f_t^2 \cdots f_t^F]$, here $f \in C$ and F denotes the length of the call stack, $t \in [0, T]$ and T is the length of the trajectory.

Goal-Conditioned Behavioral Cloning (GCBC): [26] performs behavioral cloning using future states within the same trajectory as goals. In our setting, the goal g corresponds to a target function name (from the call stack), and the agent learns a goal-conditioned policy $\pi(a \mid s, s^{\text{code}}, g)$, where s^{code} represents the call stack at the current state.
The objective used to train the policy depends on the action space:

$$J_{GCBC}^{\text{disc}}(\pi) = \mathbb{E}_{(s,s^{\text{code}},a,g)\sim\mathcal{D}}\left[\log \pi(a \mid s, s^{\text{code}}, g)\right], \tag{3.1}$$

$$J_{GCBC}^{\text{cont}}(\pi) = \mathbb{E}_{(s,s^{\text{code}},a,g)\sim\mathcal{D}}\left[\|\pi(s, s^{\text{code}}, g) - a\|^2\right], \tag{3.2}$$

where $\mathcal{D}$ is a set of transition tuples $(s, s^{\text{code}}, a, r, s', s^{\text{code}'}, g)$, where $(s', s^{\text{code}'})$ is a code-aware state visited upon taking action a from code-aware state (s, code), and g is a goal—Dataset construction is discussed in detail in Section 6. Equations 3.1 and 3.2 correspond to the loss functions used in discrete and continuous action space, respectively.

Goal-Conditioned Implicit Q-Learning (GCIQL) Goal-conditioned variant of Implicit Q-Learning (IQL) [27], which learns value functions using expectile regression. In our setup, both the value and Q-functions are conditioned on the call stack s^{code} in addition to the state and goal. GCIQL learns $Q(s, s^{\text{code}}, a, g)$ and $V(s, s^{\text{code}}, g)$ by minimizing the following losses:

$$\mathcal{L}_V^{\text{GCIQL}}(V) = \mathbb{E}_{(s,s^{\text{code}},a,g)\sim\mathcal{D}}\left[\ell_\kappa^2\left(\bar{Q}(s, s^{\text{code}}, a, g) - V(s, s^{\text{code}}, g)\right)\right], \tag{3.3}$$

$$\mathcal{L}_Q^{\text{GCIQL}}(Q) = \mathbb{E}_{(s,s^{\text{code}},a,s',s^{\text{code}'},g)\sim\mathcal{D}}\left[\left(r(s, s^{\text{code}}, g) + \gamma V(s', s^{\text{code}'}, g) - Q(s, s^{\text{code}}, a, g)\right)^2\right], \tag{3.4}$$

where $r(s, s^{\text{code}}, g) = \mathbb{I}\{g \in s^{\text{code}}\} - 1$ is the sparse goal-conditioned reward function, $\bar{Q}$ is the target Q-function, and ℓ_κ^2 is the expectile loss with expectile parameter κ, which is set to 0.8. To extract a policy from the learned value functions, we use a behavior-regularized

actor update based on TD3+BC [28]:

$$J_{\text{TD3+BC}}(\pi) = \mathbb{E}_{(s,s^{\text{code}},a,s',s^{\text{code}'},g)\sim\mathcal{D}}\left[Q(s, s^{\text{code}}, \pi(s, s^{\text{code}}, g), g) - \lambda\ J_{GCBC}(\pi)\right], \quad (3.5)$$

where λ controls the trade-off between Q-value maximization and staying close to the behavior policy using BC loss.

Decision transformers: The decision transformer (DT) architecture was proposed by Lili et al. [29] to efficiently handle the sequence modeling problem. DT outputs the optimal actions by conditioning an autoregressive model on the desired return (reward), past states, and action. The simplicity and scalability of the model allow us to encode multi-modal data. For our work, it's the observation and call stack. The **Transformer** architecture introduced by [30], are powerful model designed to handle sequential data efficiently. The core idea behind Transformers is the self-attention mechanism, which allows the model to focus on different parts of the input sequence when making predictions. A Transformer is built using multiple layers of self-attention and feedforward networks, each with residual connections to help with training. In each self-attention layer, the input sequence is first converted into a set of n embeddings $\{x_i\}_{i=1}^n$, one for each token. These embeddings are then transformed into queries q_i, keys k_i, and values v_i through learned linear projections. The output embedding z_i for the i-th token is computed as a weighted sum of all value vectors, where the weights are determined by the similarity between the i-th query and each key:

$$z_i = \sum_{j=1}^{n} \text{softmax}\left(\{\langle q_i, k_{j'}\rangle\}_{j'=1}^{n}\right)_j \cdot v_j, \quad (3.6)$$

This mechanism allows the model to learn which parts of the input are most relevant to each other, making Transformers especially effective for tasks involving long-range dependencies.

4. METHOD

In this section, we describe our core contribution: augmenting goal-conditioned reinforcement learning agents with call stack information. We present 1) Call Stack encoding and 2) our adaptation of the Decision Transformer [29] for automated gameplay testing. Unlike the original framework that conditions on scalar rewards or return-to-go, our formulation conditions the policy on a desired goal and incorporates call stack traces at each timestep. This setup shifts the focus from optimizing for cumulative rewards to achieving programmatic objectives, making it well-suited for test generation tasks. The overall architecture is illustrated in Figure 2, and Algorithm 1 highlights differences compared to the original Decision Transformer (DT) algorithm [29].

4.1. CALL STACK ENCODING

At each timestep t, we include the call stack s_t^{code} as an ordered list of function calls. We experiment with three different approaches to encode s_t^{code}, which consists of an ordered list of function $[f^1, f^2 \cdots f^F]$:

- **Simple Token Embedding**: Each function in s^{code} is embedded via an embedding table[1] $\mathcal{E} : C \to \mathbb{R}^d$, where d denotes the embedding dimension. The resulting vectors are summed to form a single embedding for the call stack.

$$\text{CS}_{emb} = \sum_{i=1}^{F} \mathcal{E}(f^i) \quad (4.1)$$

[1] A simple lookup table that stores embeddings of a fixed dictionary and size

- **Linear Projection**: We obtain function embeddings from the embedding table $\mathcal{E}$, apply a learned linear projection $W \in \mathbb{R}^{d \times d}$ to each embedding in the sequence, and then sum the resulting vectors:

$$\mathrm{CS}_{emb} = \sum_{i=1}^{F} W \cdot \mathcal{E}(f^i) \tag{4.2}$$

- **Multi-Head Self-Attention (MHSA)**: Function embeddings are processed using a Multi-Head Self-Attention (MHSA) block [30]. The resulting output tokens are then aggregated by summation into a single context vector:

$$\mathrm{CS}_{\mathrm{emb}} = \sum_{i=1}^{F} \mathrm{MHSA}([\mathcal{E}(f^1), \mathcal{E}(f^2), \ldots, \mathcal{E}(f^F)])_i \tag{4.3}$$

In the case of DT CS_{emb} is concatenated with other token embeddings, as shown in Figure 2. Whereas in GCBC and GCIQL, CS_{emb} is concatenated along with the goal and the state representation. We find that using a transformer-based encoder for the call stack leads to improved generalization and executing desired test behaviors.

4.2. **AGENT: GC-DT**

Algorithm 1: Goal Conditioned DT with Call Stack

```
# g, cs, s, a, t: goal, call stack, state, action, timestep
# transformer: GPT Style
# embed_g, embed_a: Linear Embedding Layers
# embed_s: Conv / Linear Embedding Layers
# encode_cs: encoder for call stack
# embed_t: learned timestep embedding
# pred_a: action prediction head

# main model
def CodeAwareAgent(g, cs, s, a, t):
  # shared timestep embedding
  pos_emb = embed_t(t)

  # compute embeddings
  g_emb = embed_g(g) +  pos_emb
  # one token per call stack
  cs_emb = encode_cs(cs) + pos_emb
  s_emb = embed_s(s) + pos_emb
  a_emb = embed_a(a) + pos_emb

  # interleave input as
  # (g, cs_1, s_1, a_1, ..., cs_K, s_K, a_K)
  input_emb = stack(g_emb, cs_emb, s_emb, a_emb)

  # pass through transformer
  hidden_states = transformer(input_emb=input_emb)

  # select action token embeddings for prediction
  a_hidden = unstack(hidden_states).actions

  # predict action
  return pred_a(a_hidden)
```

Our goal is to model behavior that leads to the execution of specific functions, which are used as goals during training and inference. To support goal-conditioned generation, we represent trajectories using a sequence of tokenized inputs that include the desired goal g, the call stack s_t^{code} at each timestep, the environment state s_t, and the action a_t taken.

This leads to the following autoregressive trajectory format:

$$\tau = \left(g, s_1^{\mathsf{code}}, s_1, a_1, s_2^{\mathsf{code}}, s_2, a_2, \ldots, s_T^{\mathsf{code}}, s_T, a_T\right), \tag{4.4}$$

where g is fixed for a given trajectory and is prepended to the sequence as conditioning. Unlike standard Decision Transformers that condition on returns-to-go, we explicitly provide a function-level goal g that corresponds to a code path we want the agent to trigger. This enables the model to generate actions that steer the agent toward desired programmatic behavior, rather than achieving numerical rewards.

Training and inference. During training, the model learns to predict the next action given the goal, call stack, and state:

$$\pi(a_t \mid g, s_{\leq t}^{\mathsf{code}}, s_{\leq t}, a_{<t}). \tag{4.5}$$

At test time, the agent is given a goal g and the initial environment state. The model generates actions autoregressively, using the evolving call stack and state history as context. This allows the agent to repeatedly test whether specific functionality is covered under different gameplay conditions.

Architecture. We feed the last K timesteps into our Goal-Conditioned Decision Transformer, resulting in a sequence of $4K$ tokens: one for the goal, call stack, state, and action. Each input modality is embedded using a learned projection: we apply a linear layer to the state and action, and a Multi-Head Self-Attention encoder for the call stack. This is followed by layer normalization [31] to stabilize training. For environments with visual observations, such as crafter, the state is first processed through a convolutional encoder, which outputs a latent vector before projection into the token embedding space. Additionally, each timestep is assigned a learned timestep embedding, which is added to all tokens corresponding to that timestep.

5. ENVIRONMENT

We built two environments that support RL training and code profiling. These environments are built on top of Godot [9] and Crafter [10].

5.1. GODOT

Godot is an open-source game engine, and Godot RL Agent [9] is customized for Deep Reinforcement Learning (DRL) research. It provides an interface between the game engine and the RL algorithm. In Godot Multilevel Robot, the bot has to start from an initial location and must navigate to higher levels. To enable code-based information, we manually modified the game engine code to incorporate a code state (call stack), along with the usual observations. The agent receives feedback on the corresponding function calls with each action taken in the environment. The observation space is a 1D vector that includes Raycast data relative to the player and the distance to collectibles. In addition to this, we also append the function calls triggered by the action that led to this observation. We identified 34 gameplay-related functions that are called at each action step. The environment uses an action repeat of 8. To reduce redundancy, we exclude functions that are executed every frame regardless of the agent's actions. After eliminating the unnecessary functions, we ended up with 24 interesting functions that can be potential goals for the RL agents.

5.2. CRAFTER

Crafter is an open-world survival game with pixel-based observation that evaluates a wide range of general abilities within a single environment. There were a total of 293

functions related to the gameplay that were being tracked for each action step. Similar to the filtering performed on Godot, we eliminated functions that were invoked at every action step, regardless of the agent's behavior. This left us with 94 distinct functions. These functions represent meaningful gameplay events and can serve as potential goals for regression testing using RL agents. The state space consists of an RGB image and the function calls triggered by the action that led to this observation. We kept the instrumentation minimal to make it easy for others to use. Instead of adding changes that are only useful for game testing. Further details can be found in the Appendix.

5.3. DATA COLLECTION

For this work, we collect data by manually playing the game to emulate the real game testing scenario, where we have a human tester collecting player data. For Crafter, we collected 300 trajectories with approximately 100k samples, and for Godot, which was a relatively simpler game compared to Crafter. We stopped at around 20k samples, 200 trajectories.

6. EXPERIMENTS

We divide our experiments into two main parts. The primary focus is the offline setting, where the agent leverages existing gameplay data collected from human testers. In this setting, we compare Code-Aware Agent against other goal-conditioned offline reinforcement learning methods to assess function coverage and gameplay performance. Next, we present results with different design choices, demonstrating that CA2 (use of call stack information with Multi-Head Self Attention) significantly aids learning. We compare it with simple encoding techniques. Finally, we also conduct an ablation study to validate the effectiveness of our design choices that are meaningful to the agent's performance.

6.1. EVALUATION SETUP

All reported values are averaged over 5 random seeds, with the best-performing model per seed selected based on the maximum success rate. To ensure fairness, all algorithms are trained with the same number of gradient updates and evaluated under identical conditions. During Evaluation, scores are averaged across 10 episodes for each target function. In each episode, the agent is provided with a specific function-level goal and is evaluated on its ability to reach this goal. Importantly, the episode continues even after the goal is reached, allowing the agent to interact with the environment further. This enables comprehensive measurement of episodic return and function-level coverage. Function coverage is computed after all evaluation episodes have been completed. It reflects whether the agent has successfully triggered various functions at least once throughout the entire evaluation set, not necessarily in every episode. This metric captures the breadth of the agent's exploratory and functional behaviors, complementing the goal-specific success rate without being redundant with it. Notably, high function coverage may occur even when the success rate is lower, indicating that the agent is still visiting and exercising meaningful parts of the codebase, which is beneficial for robust testing.

6.2. EVALUATION METRICS

In our offline experiments, we evaluate agent performance using three key metrics:

(1) **Success Rate:** This is the primary metric, defined as the percentage of test-time episodes in which the agent successfully reaches the target function specified by the goal. An episode is considered successful if the agent reaches the desired function

within the allowed trajectory length. For each goal, we run 10 episodes and report the average success rate.

(2) **Episode Return:** The cumulative reward collected during an episode. This reflects how effectively the agent performs within the environment, according to the task-specific reward structure.

(3) **Function Coverage:** The number of unique functions triggered by the agent during an evaluation episode. This metric captures how extensively the agent interacts with the game environment and provides insight into its behavioral diversity.

Together, the success rate serves as a direct indicator of goal achievement, while function coverage and reward provide complementary insights into the agent's behavior.

7. RESULTS

In this section, we compare the performance of Code-Aware Agent in offline settings across two game environments, using 3 different metrics. We will also try to understand the reason behind these results in detail.

7.1. OFFLINE RESULTS

The results in Table 1 show that Code-Aware Agent significantly improves both task performance and function coverage compared to standard behavior cloning (BC) and goal-conditioned (GC) baselines. Specifically, GC agents equipped with code awareness consistently outperform their non-code-aware counterparts across all metrics. This highlights that incorporating structured program signals like the call stack helps the agent not only achieve a better success rate but also explore the functional space of the environment more effectively without losing the ability to play the game. Notably, the GC-DT + CA2 model achieves the highest success rates and function coverage, indicating that it can both accomplish game objectives and visit meaningful states. This aligns well with the idea of regression testing, where the agent must test key functions to verify behavioral correctness, suggesting that our method is capable of understanding and replaying functional structure in the environment. Among the GC methods, GC-DT shows superior performance compared to GC-IQL and GC-BC. This improvement can be attributed to the use of a context window, which allows the model to reason over longer temporal horizons and condition more effectively on the target function. Overall, GC agents with code awareness demonstrate superior generalization and goal-following ability over simple BC and goal-agnostic baselines.

		GC-BC		GC-IQL		GC-DT		BC
		Simple	CA2	Simple	CA2	Simple	CA2	
Success Rate	Crafter	39.67 ± 0.52	$\mathbf{42.65 \pm 0.72}$	41.78 ± 0.60	$\mathbf{43.28 \pm 0.63}$	48.78 ± 0.35	$\mathbf{56.47 \pm 0.49}$	-
	Godot	25.45 ± 0.29	$\mathbf{26.93 \pm 0.50}$	23.88 ± 0.52	$\mathbf{25.56 \pm 0.53}$	41.22 ± 0.54	$\mathbf{47.15 \pm 0.77}$	-
Episode Return	Crafter	1.62 ± 0.16	$\mathbf{2.18 \pm 0.15}$	3.63 ± 0.18	$\mathbf{4.08 \pm 0.19}$	4.90 ± 0.58	$\mathbf{5.12 \pm 0.60}$	1.65 ± 0.31
	Godot	$\mathbf{8.17 \pm 0.39}$	8.12 ± 0.79	$\mathbf{8.13 \pm 0.37}$	8.08 ± 0.19	13.12 ± 0.59	$\mathbf{16.15 \pm 1.12}$	7.90 ± 1.51
Function Coverage	Crafter	81.24 ± 0.42	$\mathbf{81.93 \pm 0.37}$	80.74 ± 0.72	$\mathbf{84.20 \pm 0.63}$	92.96 ± 0.00	$\mathbf{96.92 \pm 0.00}$	79.44 ± 2.13
	Godot	$\mathbf{61.23 \pm 0.47}$	61.18 ± 0.68	60.97 ± 0.37	$\mathbf{61.86 \pm 0.55}$	71.96 ± 0.24	$\mathbf{79.49 \pm 1.88}$	60.10 ± 1.12

Table 1. Performance of various offline agents across *Crafter* and *Godot*, evaluated using three metrics: Success Rate, Episode Return, and Function Coverage. "Simple" refers to non-code-aware agents, while "CA2 " denotes code-aware variant. Here, the BC agent is not conditioned on the goal function, hence the corresponding Success Rates are empty.

7.2. EFFECT OF CONTEXT LENGTH

We study the impact of context length in the decision transformer (Table 2). This choice of L was decided based on the episode length corresponding to each environment. Without CA2, performance saturates around a context of 10-20. With CA2, longer contexts (20–30) lead to better performance, particularly on Godot, where the success rate jumps from 30% to 47%. This shows that code-aware agents need more context to understand longer execution histories and make better decisions over time.

		GC-DT					
		L		2L		3L	
		Simple	CA2	Simple	CA2	Simple	CA2
Success Rate	Crafter	$48.62, \pm 0.64$	$\mathbf{55.62 \pm 1.33}$	48.71 ± 0.66	$\mathbf{56.60 \pm 0.53}$	48.78 ± 0.35	$\mathbf{56.47 \pm 0.49}$
	Godot	26.67 ± 0.95	$\mathbf{30.90 \pm 0.76}$	36.41 ± 0.85	$\mathbf{40.14 \pm 2.04}$	41.22 ± 0.88	$\mathbf{47.15 \pm 0.77}$

Table 2. This table summarizes the performance of Decision Transformers across different context lengths. For Crafter, $L = 10$, while for Godot $L = 5$.

7.3. DESIGN CHOICE

Finally, we evaluate different architectural designs for integrating call stack information (Table 3) through an offline ablation study, using success rate as the primary metric. We find that a simple embedding of the call stack performs poorly. While adding a Multi-Head Self Attention (MHSA)[2] layer significantly improves performance, achieving the highest success rates on both Crafter (56%) and Godot (47%). These results highlight the importance of capturing dependencies between call stack elements, something MHSA is particularly well-suited for in our setting.

	GC-DT (CA2)		
	Emb	Linear	MHSA
Crafter	47.63 ± 1.28	49.15 ± 0.87	$\mathbf{56.90 \pm 1.01}$
Godot	34.53 ± 1.46	35.15 ± 1.16	$\mathbf{39.40 \pm 6.78}$

Table 3. This table summarizes the Success Rate of different Encoder choices. (Results averaged across all three context lengths)

8. CONCLUSION

In this work, we show that our Causal Auto-regressive Architecture improves agent performance, particularly when combined with longer context and self-attention. Our results demonstrate that incorporating code-level signals, such as call-stack information, helps agents better understand how their actions relate to the game logic, leading to more structured and effective testing.

More broadly, this approach moves toward automated, code-aware game testing that reduces reliance on manual playtesting. While scalability remains a challenge due to the large size of modern game codebases, a modular testing strategy—where agents focus on specific features or submodules can make the system more practical. Integrating code-aware agents early in development could enable faster iteration, targeted debugging, and more reliable validation of game components.

[2]GC-DT + MHSA is our best Code-Aware Agent (CA2)

9. LIMITATION AND FUTURE WORK

A key limitation of our approach is the high computational cost of collecting real-time function-level code coverage and call stack information in complex game environments. This instrumentation introduces runtime overhead and limits scalability to larger games with many functions and events. Moreover, treating all functions equally may not reflect practical developer priorities. In future work, we will focus on smaller, modular environments and explore selective instrumentation that tracks only developer-specified or high-priority functions. This would reduce overhead, improve practical relevance, and make the framework more scalable for real-world game development.

ACKNOWLEDGMENT

We would like to thank Sarra Habchi, Ian Guak, Alessandro Palmas, and Gabriel Robert for their valuable discussions and feedback throughout this project. We also thank Harley Wiltzer and Mahtab (Mattie) Nejati for their helpful comments on the manuscript. This work was supported by the Mitacs PhD Accelerate Grant. This research was enabled in part by compute resources, software, and technical support provided by Ubisoft (La Forge).

References

[1] C. Politowski, F. Petrillo, and Y.-G. Guéhéneuc. *A Survey of Video Game Testing*. 2021. arXiv: 2103.06431 [cs.SE]. URL: https://arxiv.org/abs/2103.06431.

[2] J. Gillberg, J. Bergdahl, A. Sestini, A. Eakins, and L. Gisslén. "Technical Challenges of Deploying Reinforcement Learning Agents for Game Testing in AAA Games". In: *2023 IEEE Conference on Games (CoG)*. 2023. DOI: 10.1109/CoG57401.2023.10333194.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[4] D. Hafner, J. Pasukonis, J. Ba, and T. Lillicrap. *Mastering Diverse Domains through World Models*. 2024. arXiv: 2301.04104 [cs.AI]. URL: https://arxiv.org/abs/2301.04104.

[5] C. Gordillo, J. Bergdahl, K. Tollmar, and L. Gisslén. *Improving Playtesting Coverage via Curiosity Driven Reinforcement Learning Agents*. 2021. arXiv: 2103.13798 [cs.LG]. URL: https://arxiv.org/abs/2103.13798.

[6] A. Sestini, L. Gisslén, J. Bergdahl, K. Tollmar, and A. D. Bagdanov. "Automated Gameplay Testing and Validation With Curiosity-Conditioned Proximal Trajectories". In: *IEEE Transactions on Games* 16.1 (2024), pp. 113–126. DOI: 10.1109/TG.2022.3226910.

[7] G. Liu, M. Cai, L. Zhao, T. Qin, A. Brown, J. Bischoff, and T.-Y. Liu. *Inspector: Pixel-Based Automated Game Testing via Exploration, Detection, and Investigation*. 2022. arXiv: 2207.08379 [cs.AI]. URL: https://arxiv.org/abs/2207.08379.

[8] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. "Openai gym". In: *arXiv preprint arXiv:1606.01540* (2016).

[9] E. Beeching, J. Debangoye, O. Simonin, and C. Wolf. *Godot Reinforcement Learning Agents*. 2021. arXiv: 2112.03636 [cs.LG]. URL: https://arxiv.org/abs/2112.03636.

[10] D. Hafner. *Benchmarking the Spectrum of Agent Capabilities*. 2022. arXiv: 2109.06780 [cs.AI]. URL: https://arxiv.org/abs/2109.06780.

[11] K. Chen, Y. Li, Y. Chen, C. Fan, Z. Hu, and W. Yang. "GLIB: towards automated test oracle for graphically-rich applications". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE '21. ACM, Aug. 2021, 1093–1104. DOI: 10.1145/3468264.3468586. URL: http://dx.doi.org/10.1145/3468264.3468586.

[12] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, et al. "Grandmaster level in StarCraft II using multi-agent reinforcement learning". In: *nature* 575.7782 (2019), pp. 350–354.

[13] S. Lifshitz, K. Paster, H. Chan, J. Ba, and S. McIlraith. *STEVE-1: A Generative Model for Text-to-Behavior in Minecraft*. 2024. arXiv: 2306.00937 [cs.AI]. URL: https://arxiv.org/abs/2306.00937.

[14] S. Qi, S. Chen, Y. Li, X. Kong, J. Wang, B. Yang, P. Wong, Y. Zhong, X. Zhang, Z. Zhang, N. Liu, W. Wang, Y. Yang, and S.-C. Zhu. *CivRealm: A Learning and Reasoning Odyssey in Civilization for Decision-Making Agents*. 2024. arXiv: 2401.10568 [cs.AI]. URL: https://arxiv.org/abs/2401.10568.

[15] B. Baker, I. Akkaya, P. Zhokhov, J. Huizinga, J. Tang, A. Ecoffet, B. Houghton, R. Sampedro, and J. Clune. *Video PreTraining (VPT): Learning to Act by Watching Unlabeled Online Videos*. 2022. arXiv: 2206.11795 [cs.LG]. URL: https://arxiv.org/abs/2206.11795.

[16] S. Team et al. *Scaling Instructable Agents Across Many Simulated Worlds*. 2024. arXiv: 2404.10179 [cs.RO]. URL: https://arxiv.org/abs/2404.10179.

[17] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar. *Voyager: An Open-Ended Embodied Agent with Large Language Models*. 2023. arXiv: 2305.16291 [cs.AI]. URL: https://arxiv.org/abs/2305.16291.

[18] J. S. Park, J. C. O'Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein. *Generative Agents: Interactive Simulacra of Human Behavior*. 2023. arXiv: 2304.03442 [cs.HC]. URL: https://arxiv.org/abs/2304.03442.

[19] C. Wang, L. Tang, M. Yuan, J. Yu, X. Xie, and J. Bu. "Leveraging LLM Agents for Automated Video Game Testing". In: *arXiv preprint arXiv:2509.22170* (2025).

[20] Y. Qin et al. *UI-TARS: Pioneering Automated GUI Interaction with Native Agents*. 2025. arXiv: 2501.12326 [cs.AI]. URL: https://arxiv.org/abs/2501.12326.

[21] Y. Zheng, X. Xie, T. Su, L. Ma, J. Hao, Z. Meng, Y. Liu, R. Shen, Y. Chen, and C. Fan. "Wuji: Automatic Online Combat Game Testing Using Evolutionary Deep Reinforcement Learning". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019, pp. 772–784. DOI: 10.1109/ASE.2019.00077.

[22] S. Agarwal, C. Herrmann, G. Wallner, and F. Beck. "Visualizing AI Playtesting Data of 2D Side-scrolling Games". In: *2020 IEEE Conference on Games (CoG)*. 2020, pp. 572–575. DOI: 10.1109/CoG47356.2020.9231915.

[23] J. Bergdahl, C. Gordillo, K. Tollmar, and L. Gisslén. *Augmenting Automated Game Testing with Deep Reinforcement Learning*. 2021. arXiv: 2103.15819 [cs.LG]. URL: https://arxiv.org/abs/2103.15819.

[24] P. Le Pelletier de Woillemont, R. Labory, and V. Corruble. "Automated Play-Testing through RL Based Human-Like Play-Styles Generation". In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 18.1 (Oct. 2022), 146–154. ISSN: 2326-909X. DOI: 10.1609/aiide.v18i1.21958. URL: http://dx.doi.org/10.1609/aiide.v18i1.21958.

[25] P. V. Amadori, T. Bradley, R. Spick, and G. Moss. *Robust Imitation Learning for Automated Game Testing*. 2024. arXiv: 2401.04572 [cs.LG]. URL: https://arxiv.org/abs/2401.04572.

[26] C. Lynch, M. Khansari, T. Xiao, V. Kumar, J. Tompson, S. Levine, and P. Sermanet. *Learning Latent Plans from Play*. 2019. arXiv: 1903.01973 [cs.RO]. URL: https://arxiv.org/abs/1903.01973.

[27] I. Kostrikov, A. Nair, and S. Levine. *Offline Reinforcement Learning with Implicit Q-Learning*. 2021. arXiv: 2110.06169 [cs.LG]. URL: https://arxiv.org/abs/2110.06169.

[28] S. Fujimoto and S. S. Gu. *A Minimalist Approach to Offline Reinforcement Learning*. 2021. arXiv: 2106.06860 [cs.LG]. URL: https://arxiv.org/abs/2106.06860.

[29] L. Chen, K. Lu, A. Rajeswaran, K. Lee, A. Grover, M. Laskin, P. Abbeel, A. Srinivas, and I. Mordatch. "Decision transformer: Reinforcement learning via sequence modeling". In: *Advances in neural information processing systems* 34 (2021), pp. 15084–15097.

[30] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[31] J. L. Ba, J. R. Kiros, and G. E. Hinton. *Layer Normalization*. 2016. arXiv: 1607.06450 [stat.ML]. URL: https://arxiv.org/abs/1607.06450.

Appendix A. CA2 ADDITIONAL DETAILS

A.1. HYPERPARAMETERS

We present the hyperparameters (Table 5) used for training our Decision Transformer with Code-Aware Agent (GC-DT-CA2) agent. This configuration reflects design choices tailored to effectively capture temporal dependencies, program structure, and goal representations in structured game environments. We detail model architecture parameters such as embedding dimensions, transformer depth, and attention heads, along with training settings including optimizer parameters, learning rate scheduler, and dropout.

A.2. ENVIRONMENT DETAILS

Table 4. Environment Details for Crafter and Godot

Attribute	**Crafter**	**Godot**
State Type	Image	Vector
State Dimension (s)	$64 \times 64 \times 3$	65×1
Action Space	Discrete [0, 16]	Continuous [-1,1]
Action Dimension (a)	17	2
Number of Functions (f)	293	34
Number of Goals (g)	94	24
Code Trace (s^{code})	$[f^1, f^2, \ldots, f^F]$	$[f^1, f^2, \ldots, f^F]$
Function calls per step (F)	$[0, 600]$	$[0, 15]$

Table 4 provides environment specific details. The "Function calls per step"F indicates the total number of functions triggered by a single action. In Crafter, this number is relatively high, especially at the start of the game due to initialization, though it typically comes down around 100–200 function calls per step on average. In contrast, the number is significantly lower in Godot, primarily due to limited access to instrumentation within the Godot engine codebase.

We present the hyperparameters (Table 5) used for training our Decision Transformer with Code-Aware Agent (GC-DT-CA2) agent. This configuration reflects design choices tailored to effectively capture temporal dependencies, program structure, and goal representations in structured game environments. We detail model architecture parameters such as embedding dimensions, transformer depth, and attention heads, along with training settings including optimizer parameters, learning rate scheduler, and dropout.

A.3. LATENCY

Our implementation uses lightweight instrumentation that logs active function identifiers directly from the engine's call stack interface, adding minimal overhead. We are currently profiling the Environment latency time per environment step with and without code instrumentation. Experiments show that instrumentation adds less than 5–8% overhead in both Crafter and Godot environments, negligible relative to the total simulation time, suggesting that the overhead is dominated by environment simulation rather than by the instrumentation itself.

A.4. COMPUTE REQUIREMENTS

All experiments were conducted on a machine equipped with an NVIDIA A4000 GPU and 16 GB of memory. The peak memory usage for the CA2 agent was approximately $\sim 13.6GB$, while the simple agent required around $\sim 4GB$.

Table 5. Decision Transformer Hyperparameters

Parameter	**Value / Description**
General Settings	
Agent Type	`Simple` / `cs`
Batch Size	128
Epochs	10
Training Steps	10000
Evaluation Episodes	10
GPT Model Configuration	
Context Length	L / 2L / 3L
Value of L	5 (Godot) / 10 (Crafter)
Token Dimension	128
Transformer Layers (n_layer)	2
Attention Heads (n_head)	2
Dropout	0.1
Learning Rate	3×10^{-4}
Adam Betas	(0.9, 0.95)
Gradient Norm Clip	1.0
Weight Decay	0.1
LR Scheduler	Linear warmup + cosine decay
Image Encoder (Crafter)	
Encoder Channels	32, 64, 64
Encoder Filter Sizes	8×8, 4×4, 3×3
Encoder Strides	4, 2, 1
State Encoder (Godot)	
Number of Linear Layer	2
Hidden Dimension	1024
Call Stack Encoder: (MHSA) 4.1	
Function Embedding Dimension	128
Goal Embedding Dimension	128
Number of Layers	1
Number of heads	2

A.5. **CODE**

The code for the CA2 agent will be made publicly available on acceptance.