

Autopilot: A Continually Learning Real-Time General Playtesting System

Nyx Iskandar, Andy Tsen

Ramen team, the developers behind Aura

www.tryaura.dev · [@Aura_Assistant](https://twitter.com/Aura_Assistant)

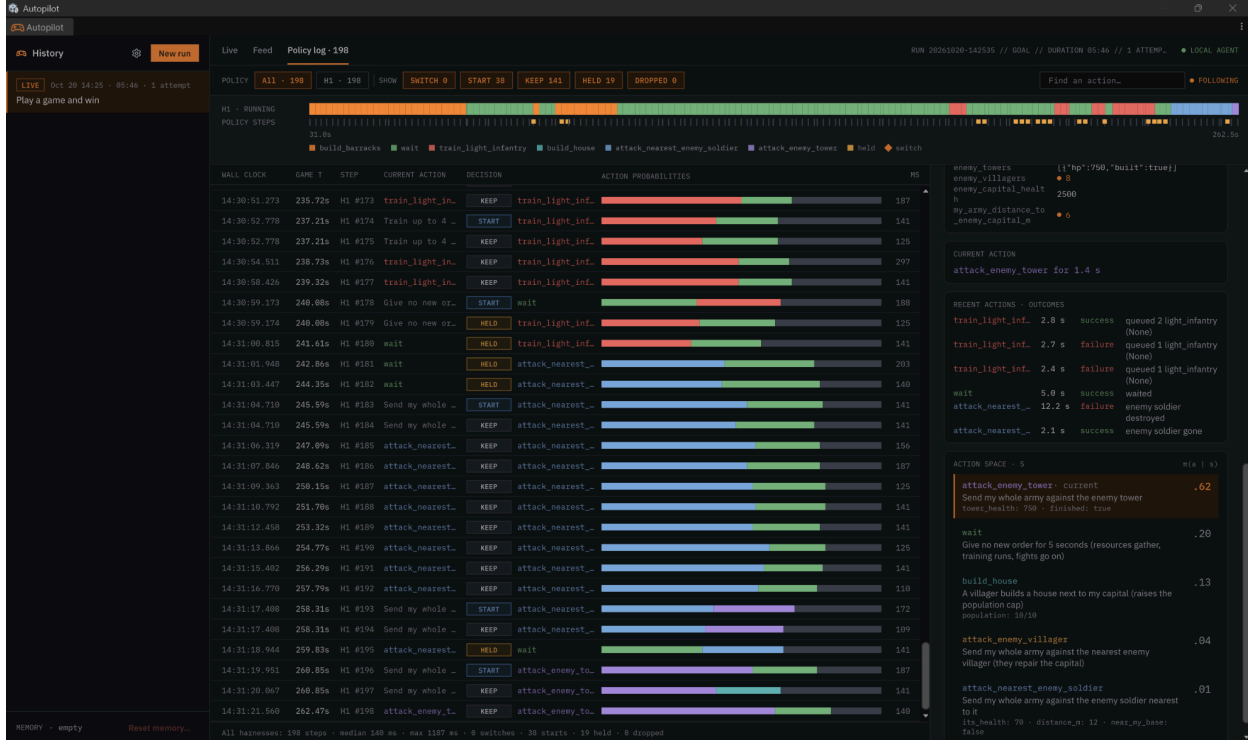
Autopilot is an agentic in-editor playtesting system of large language and decision models designed for generality across games, game engines, operating systems, and playtesting tasks. Compatible with Unreal Engine 5, Unity 6, and Godot 4 on Windows and MacOS. Autopilot has been evaluated against and proven to successfully play different game genres using real user input events, including real-time strategy, first-person shooters, survival crafting, turn-based role-playing, match-3, and Soulslike games. This generality is achieved through Autopilot’s three key features: low-latency architecture for real-time decision models; adaptive gameplaying harness generation where each harness is written dynamically yet bespoke to each particular game; and per-game neurosymbolic continual learning.

GitHub: <https://github.com/ramenvr/autopilot/>

1 Introduction

Game AI is not a new concept, predating agents built on large language models (LLMs). Traditional game AI leverages decision trees, explicit rules which control how the agent would act given a certain game state. In the 2010s and early 2020s, progress in reinforcement learning was largely measured against game benchmarks like the Arcade Learning Environment (Atari) benchmark, making methods like AlphaGo headlines in and out of the tech world. With LLMs, the genesis of a highly capable, general game-playing agent may not be so far-fetched after all.

However, while we believe that LLMs are a necessary component for building the most capable and most efficient general game-playing agent, they are not sufficient. Our bet is that decision models are another necessary component for such an agent, improving speed, cost, and capability. In the limit, the overall latency of a decision approaches the inference latency of the decision model, and a low-latency architecture – e.g. to enable batch-reads of game state, minimize server hops between the client and the model provider – is a necessary ingredient for this.



Actions taken by the decision model displayed in Autopilot's Unreal Engine plugin UI.

One can argue that a general game-playing agent is a gimmick, and we agree! It doesn't take a game designer to know that games are built for human entertainment. So why did we build a general game-playing agent?

Quality Assurance. One of the biggest challenges in creating a game is functional testing. This is the process in which QA testers search for bugs in the level, verify that a particular game state is reachable, etc. As game developers ourselves, we know that functional testing is an expensive, time-consuming, and repetitive process that takes away the fun in playing a game.

Undoubtedly, a playtester must be able to competently play a game, any game, and be fast but thorough in their assessment. Accordingly, the goal of building a general playtesting system is our motivation for building a general game-playing agent. And this is how we arrived at Autopilot.

Autopilot

History

CONFIRMED

Oct 20 16:08 · 02:45 · 1 att...

Verify that shooting the weapon shows a visible muzzle flash or impact effect.

New run

Result

Feed

Policy log · 16

RUN 20261020-161111 // VERIFY // DURATION 02:45 // 1 ATTEMPT

RUN 20261020-161111 // VERIFY // OCT 20 16:08

Verify that shooting the weapon shows a visible muzzle flash or impact effect.

Started from Claude Code

CONFIRMED

Shooting the assault rifle shows a visible muzzle flash. In episode 1 the agent fired 5 bursts in first person at the wall in front of the Team A spawn on Conquest, choosing among fire, reload, turn and aim options. Filmed at 20 fps and described frame by frame for two of the bursts, each showed a yellow-orange flame with a bright white-yellow core and sparks at the rifle's front end. These appeared in the same frames where the HUD ammo counted down and the rifle recoiled. The flame was gone in the frames after firing stopped, and the nearby wall took an orange tint while it showed. The impact effect wasn't clearly seen: the wall was about 78 cm away and the flash covered that spot, so no distinct bullet holes could be made out. The claim ("muzzle flash or impact effect") rests on the muzzle flash. Side observation: in both bursts most of the flame sat left of the sight tip, over the wall corner, rather than centred on the muzzle. This may be how the effect looks this close to a wall, or an offset of the muzzleFlash socket; it is worth a look. No direct changes were made to the game.



EVIDENCE

```

trace1.jsonl: 5 bursts (game t 36.34-37.17 ammo 20->15; 37.43-38.29 15->11; 38.53-39.49 11->6;
39.64-40.54 6->0; reload; 43.43-44.39 30->25), weapon CurrentAmmo read directly, IsUsingTPP false
(first-person mesh)
Description, episode 1, t 36.33-37.30 (12 frames): F1 black sight, no light; F2-F5 yellow-orange
sparks, a white-yellow dot and an orange flame puff at the rifle's front end, with the rifle recoiling
up-right; F6-F7 faint orange glow or specks; F8-F12 no flash. HUD ammo 20->19->18->17->16->15 over F2-
F6
Description, episode 1, t 43.31-44.46 (12 frames): F1-F2 black sight only; F3 yellow-orange flame with
sparks; F4 brightest, with a yellow-white core in an orange cloud; F5-F6 flame with grey smoke; F7 grey
haze only; F8-F12 plain sight. Wall tinted orange in F3-F7. HUD ammo 30->29->27->26->25 over F3-F6
Code: BP_BaseWeapon.LocalPlayEffects spawns SettingsData.MuzzleFlash (NS_WeaponFire_MuzzleFlash_Rifle
for DA_AssaultRifle_Settings) on MeshFPP socket muzzleFlash; PlayEffects spawns NS_ImpactConcrete plus
a bullet-hole decal at the impact point
Not seen: distinct impact marks or decals on the wall (wall about 78 cm away, behind the flash). Both
descriptions noted the flame mostly left of the sight tip in the brightest frames.

```

MEMORY · 4 runs

Reset memory...

Autopilot playing a first-person shooter and verifying a visual feature.

The Autopilot system consists of four main components:

- Engine plugin. The plugin to the game engine enables Autopilot to read game states and execute actions by injecting real user inputs during gameplay.
- Agent service. This Python process hosts the coding agent, the decision model harness runtime, a sandbox for code written by the coding agent, and per-game memory files. It communicates with the plugin via a WebSocket bridge.
- Coding agent. Autopilot supports the Agent SDKs of Anthropic (Claude Code), OpenAI (Codex), and OpenCode. The coding agent conducts research on the game and writes the harness for the decision model.
- Decision model. Autopilot supports Jev through the TypeSafe SDK. The decision model makes real-time decisions during gameplay based on its harness, including game states it perceives.

2 Early Iterations and Empirical Tests

When iterating through early versions of Autopilot, we found that the level of abstraction of the decision model harness significantly affects how well Autopilot plays a game. Early demos of decision model-based game AI playing Doom, Flappy Bird, Pac-Man, etc. each used a harness tuned specifically for each game such that each action option for the decision model is a game-specific macro-action; for example, the action list for Flappy Bird, a relatively simple game, is [“flap”, “glide”]. By design, each macro-action must correspond to a programmatic function which translates an action choice to an actual action taken in the game, through user input events like pressing the Space button.

At first glance, this game-specific approach seems to be the antithesis to a general game-playing agent. An engineer would naturally look to descend in abstraction and instead present the decision model with atomic actions, where the action list is more akin to [“press_W”, “press_A”, “press_S”, “press_D”, “click_mouse_left”, ...]. This action list is certainly more general than the one shown for Flappy Bird. However, decision models operate in natural language, which means that they are able to more accurately predict outputs when given natural language inputs rather than keypresses and mouse clicks. So what this engineer gains in perceived generality, they lose in capability, as we have empirically verified through early versions of Autopilot.

3 The Unlock: Adaptive Harnesses

Our solution is adaptive gameplaying harness generation with LLM coding agents, keeping the abstraction level at macro-actions. Since Autopilot lives in-editor, the coding agent builds each harness through reading the game’s source code and play-in-editor states, and each harness in turn reads play-in-editor states and requests actions from the decision model. Each harness inherits from a pre-defined parent harness class included in the Autopilot kit, standardizing Autopilot harnesses across different instances of Autopilot across different users, games, game engines, and operating systems.

There are advanced features crafted into the Autopilot kit that enhances gameplay. One example is navigation, in which the system lazily builds up a map with sonar to detect obstacles by projecting the player’s capsule and finding intercepts, and subsequently using A* pathfinding to find the shortest path to a target. Moreover, certain macro-actions, such as dodging an attacking enemy, require action sequences that arrive much more rapidly than possible with a decision model which in our case outputs one decision approximately every 0.25s; in these cases, reflex functions are implemented in the harness, which batch-read game states for efficiency and programmatically output user input events if the read game state fulfills the triggering condition. Like biological reflex actions, these skip the model (the “brain”) to programmatic execution.

Perhaps the most advanced feature of Autopilot is its continual learning ability, which strictly uses a per-project memory filesystem in a neurosymbolic setup. Memory files include the game’s codex, fact notes, reusable harness code, and agent traces and reports from previous runs. This enables later runs to skip researching components it has seen before, and seed themselves on a proven working harness. The memory system does not inhibit Autopilot from adapting to source code changes; in fact, it alerts the coding agent of the changed files for it to consider when writing future harnesses.

A very impressive example is Autopilot playing Soulslike. In the first few episodes, Autopilot explored different weapons and armor, and combat tactics to defeat the boss. By the fourth episode, Autopilot is able to build effective reflex actions for dodging the boss’s attacks, equip both a sword and a shield as a good offense-defense combination, and approach the middle of the boss arena both to avoid getting stuck in corners where neither it nor the boss is able to hit the other, as well as to prevent itself from going through walls and falling into the void (which is an artifact of the particular Soulslike game it played).



The Unreal Engine play-in-editor (PIE) window in which Autopilot is pursuing the goal to “Kill the boss” in Soulslike.

4 Evaluating Autopilot’s Frontier-Level Performance

To evaluate Autopilot’s performance, we ran verification and goal-driven tasks on an unmodified version of Easy Survival RPG V5 in Unreal Engine 5.8. We selected this template game as it is a relatively large playable game with multiple possible tasks, such as menu navigation, resource collection, and combat. The tasks range from simple few-step tasks (e.g., “Verify that the player’s hunger decreases over time while the player stands still”) to complex long-running tasks (e.g., “Craft a stone hatchet and use it to kill a zombie”).

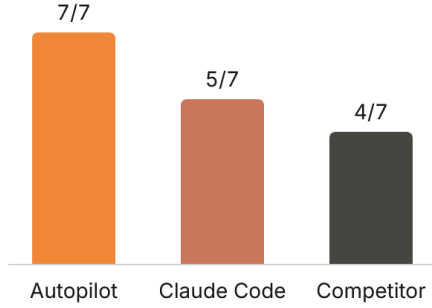
Through these evaluation runs, we aim to assess both the accuracy and speed of Autopilot with empty memory. We compare Autopilot against different harnesses, namely Claude Code and that of a leading Unreal/Unity AI agent competitor (Competitor), using Claude Opus 5.5 on medium effort. Claude Code plays through Unreal Engine’s Python Remote Execution, while Competitor plays through its generated TypeScript scripts to control the player pawn.

All tasks were run in the same Windows machine, and task duration was measured from the first generated token to the last generated token, excluding setup steps such as launching the play-in-editor window, but including time spent on research steps. The memory system was emptied prior to each run such that each run is one cold attempt at the task.

For verification tasks, we consider a result to be correct when it matches the expected verdict, which is either “confirmed” (the feature to verify is indeed implemented) or “denied” (the feature to verify is missing). For goal-driven tasks, we consider a result to be correct if it achieves the provided goal, which means that an incorrect result is one which times out or gives up before achieving the goal. We exclude exploration tasks as these tasks are open-ended in nature. We mark runs as unsuccessful if they included cheating behavior, such as teleporting the player, spawning items in the player’s inventory without collecting them in a legitimate way (e.g., collecting logs by cutting down a tree), or making the player invulnerable.

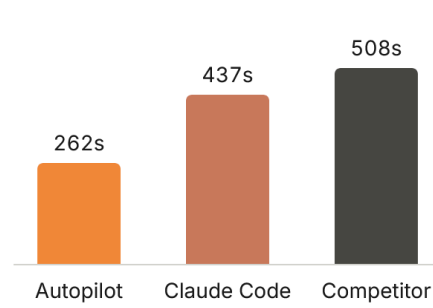
Accuracy

Pass@1, correct = goal achieved or verification is correct in under 1800s.



Mean Time Taken

Measured from first output token to last output token. Lower is better.



PROMPT	EXPECTED	Autopilot		Claude Code		Competitor	
		RESULT	TIME / s	RESULT	TIME / s	RESULT	TIME / s
Verify that the player's hunger decreases over time while the player stands still	Confirmed	✓	89	✓	303	✓	210
Verify that picking up a stone adds a stone to the player's inventory	Confirmed	✓	57	✓	331	✓	228
Check from the inventory screen whether a campfire takes 8 seconds to craft	Confirmed	✓	118	✓	156	✗	188
Check whether the crafting list shows that a campfire needs only 2 wood to craft	Denied	✓	80	✓	213	✓	127
Verify that you cannot open the inventory after starting a new game	Denied	✓	106	✓	335	✗	174
Craft a campfire and place it on the spawn island	Achieved	✓	573	✗	799	✓	828
Craft a stone hatchet and use it to kill a zombie	Achieved	✓	809	✗	924	✗	1800

The prompts written in the above tables are the exact prompts given to each agent, with the addition of “Play in PIE” at the end of each prompt for Claude Code and Competitor to avoid confusion in those agents, since they are not designed specifically for playtesting.

From observing each gameplay, Autopilot’s faster gameplay is achieved through up-front research and harness creation to leverage decision models, as opposed to Competitor’s method of repeatedly pausing the play-in-editor window to revise the LLM-generated script. On the other hand, Claude Code’s longer duration can be attributed to inefficient research token generations and tool calls. Claude Code’s goal-driven runs both contained cheating behavior, hence while the goals were achieved at the end, we marked them unsuccessful.

5 Conclusion

Autopilot unlocks a new frontier for game playtesting. We believe that this frontier can be pushed further through opening up Autopilot for open-source contributions from engineers and researchers interested in the intersection of AI, games, and infrastructure. From integrating multimodal decision models to further decreasing gameplay latency, we are excited to see what the community can build to level up Autopilot.