MIMIC-Py: An Extensible Tool for Personality-Driven Automated Game Testing with Large Language Models

Yifei Chen
yifei.chen@mail.mcgill.ca
Electrical and Computer Engineering,
McGill University
Montréal, Québec, Canada

Sarra Habchi
sarra.habchi@cohere.com
Cohere
Montréal, Québec, Canada

Lili Wei*
lili.wei@mcgill.ca
Electrical and Computer Engineering,
McGill University
Montréal, Québec, Canada

Abstract

Modern video games are complex, non-deterministic systems that are difficult to test automatically at scale. Although prior work shows that personality-driven Large Language Model (LLM) agents can improve behavioural diversity and test coverage, existing tools largely remain research prototypes and lack cross-game reusability.

This tool paper presents MIMIC-Py, a Python-based automated game-testing tool that transforms personality-driven LLM agents into a reusable and extensible framework. MIMIC-Py exposes personality traits as configurable inputs and adopts a modular architecture that decouples planning, execution, and memory from game-specific logic. It supports multiple interaction mechanisms, enabling agents to interact with games via exposed APIs or synthesized code. We describe the design of MIMIC-Py and show how it enables deployment to new game environments with minimal engineering effort, bridging the gap between research prototypes and practical automated game testing.

The source code and a demo video are available on our project webpage: https://mimic-persona.github.io/MIMIC-Py-Home-Page/.

CCS Concepts

• **Software and its engineering → Maintaining software**.

Keywords

Artificial Intelligence, Human-Like Gaming Agents, Personality-Driven Gaming Agents, Automated Game Testing, Large Language Models (LLMs)

ACM Reference Format:
Yifei Chen, Sarra Habchi, and Lili Wei. 2026. MIMIC-Py: An Extensible Tool for Personality-Driven Automated Game Testing with Large Language Models. In *Companion Proceedings of the 34th ACM Symposium on the Foundations of Software Engineering (FSE '26), June 5–9, 2026, Montreal, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

*Lili Wei is the corresponding author.

FSE Companion '26, Montréal, QC, Canada

ACM ISBN 978-1-4503-XXXX-X/2026/06
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Motivation. Modern video games have become one of the most significant sectors of the entertainment industry [13], making the maintenance of software quality through rigorous testing increasingly critical [9]. However, the complexity of modern games, characterized by their rich state spaces and non-deterministic environments, poses substantial challenges for automated testing techniques. As a result, most game studios still rely heavily on manual testing, which is costly, time-consuming, and difficult to scale.

Related Work. Prior work has explored agent-based game testing using Machine Learning (ML) techniques such as Reinforcement Learning (RL) and Imitation Learning (IL) to reduce human effort [1, 17]. However, RL-based methods depend on well-engineered reward functions and IL-based methods rely on expert demonstrations, making both require substantial manual effort to adapt to new games, tasks, or evolving game versions [11], limiting their practicality as testing tools.

More recently, Large Language Models (LLMs) have been applied to game-playing agents and demonstrated stronger adaptability with less human effort across diverse game environments [7, 14, 15, 23, 25, 29]. Despite these advances, relatively little work has explored the use of LLM-based agents as practical game-testing tools, particularly with respect to deployability and extensibility.

A further limitation shared by both ML-based agents and LLM-based agents is that they overlook behavioural diversity. Human players often adopt different strategies for similar tasks, shaped by individual personality traits [16]. However, most existing agents exhibit homogeneous, repetitive behaviour, limiting their ability to thoroughly explore game states and reducing the effectiveness of automated testing for complex game environments.

Our Contribution. To address these challenges, we previously proposed MIMIC [2], an LLM-based framework that integrates gameplay personality traits to generate diverse solutions for similar in-game tasks and achieve broader coverage. This design is grounded in empirical studies showing strong correlations between player personality traits and in-game behaviour [24, 26]. By embedding personality-driven decision-making into game-playing agents, MIMIC systematically explores diverse gameplay strategies.

In our prior evaluation, MIMIC demonstrated effectiveness across multiple games of varying scale, including *Dungeon Adventures*[22], *Shattered Pixel Dungeon*[6], and *Minecraft*[5]. In larger-scale settings, MIMIC consistently outperformed random-based baselines by up to **1.30**× in branch coverage and **14.46**× in interaction-level coverage. When evaluated in *Minecraft* against the state-of-the-art agent, ODYSSEY [12], MIMIC solved more complex, multi-step

tasks, while exhibiting substantially greater behavioural diversity. These results suggest that integrating personality-driven behaviours enhances both problem-solving capability and exploration effectiveness in automated game testing. Full experimental details are reported in our prior work [2].

Novelty of This Tool Demonstration Paper. Porting automated test-generation tools to new games remains challenging due to heterogeneous game architectures, diverse interaction interfaces, and the lack of standardized testing APIs [18]. As a result, most existing game testing tools are tightly coupled to a single game or engine, limiting their reuse and practical adoption.

Building on the original MIMIC framework, this paper presents MIMIC-Py, a Python-based implementation designed as a reusable and extensible testing tool. The original prototype was implemented in JavaScript for research validation, offering limited scalability and was difficult to adapt beyond the evaluated games. In contrast, MIMIC-Py adopts a modular architecture that isolates game-specific logic from core agent components, enabling deployment to new games through configuration, prompt adaptation, and lightweight interaction bridges.

Importantly, MIMIC-Py preserves the original framework's ability to interact with games via both structured action plans and executable behaviors, while re-engineering these mechanisms as configurable and extensible modules suitable for practical use. By decoupling planning, action execution, and self-summarization, MIMIC-Py significantly reduces the engineering effort required to port agents to new game environments.

This paper focuses on MIMIC-Py as a practical testing tool, detailing its workflow, extensibility mechanisms, and deployment process. Detailed algorithmic design and comprehensive evaluation results are available in our prior work [2] and are therefore not repeated here. A demo video illustrating how to run MIMIC-Py across games is available on our project webpage [3].

The paper is organized as follows: Section 2 presents the design and implementation of MIMIC-Py as a reusable automated game testing tool. Section 3 describes how MIMIC-Py can be extended and deployed to new game environments. Finally, Section 4 concludes the paper and outlines future directions.

2 MIMIC-Py

Figure 1 presents an overview of MIMIC-Py, a personality-driven agent-based game testing tool designed for diverse gameplay behaviors, scalable testing, and lightweight adaptation to new game environments. The system consists of four core components: the *Planner*, *Action Executor*, *Action Summarizer*, and *Memory System*.

At runtime, MIMIC-Py operates in an iterative loop. Given a testing objective and a personality trait, the Planner generates an action plan, which the Action Executor translates into concrete interactions with the game under test. The Action Summarizer evaluates the execution outcome and records a structured summary in the Memory System. Retrieved memories then inform subsequent planning, enabling long-running and context-aware testing.

2.1 Planner

The LLM Planner generates action plans during testing and incorporates personality traits to explore diverse gameplay strategies for similar testing tasks. Such a design allows users to generate diverse interaction traces without manually defining multiple behaviours or testing strategies.

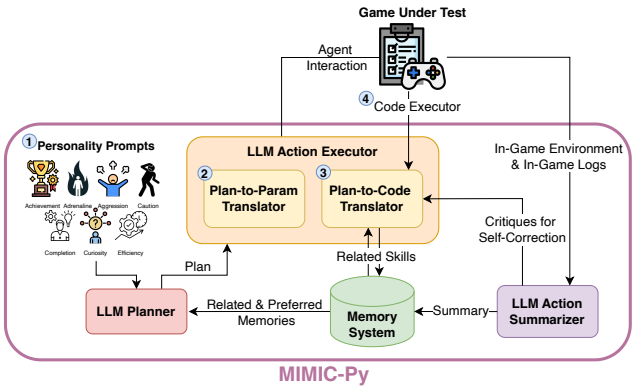

Figure 1: Overview of the MIMIC-Py framework.

To ensure MIMIC-Py can be applied to games with varying task complexity, the Planner integrates a Hybrid Planning strategy that combines Bottom-Up and Top-Down Planning. Bottom-Up Planning supports reactive, fine-grained interactions, which are effective for short-horizon or exploratory behaviours. Top-Down Planning decomposes high-level objectives into sub-tasks, supporting long-horizon reasoning and goal tracking. This Hybrid design allows MIMIC-Py to operate robustly across both exploratory and multi-step game environments without environment-specific tuning.

The Planner also conditions its decisions on the current game state and relevant past experiences retrieved from the Memory System (Section 2.4), helping maintain behavioural consistency and avoid redundant or ineffective actions across iterations.

2.1.1 ***Personality Model and Configuration.*** MIMIC-Py takes the PathOS personality model [21] to support personality-driven testing behaviours. PathOS defines seven behaviourally grounded personality traits, *Achievement*, *Adrenaline*, *Aggression*, *Caution*, *Completion*, *Curiosity*, and *Efficiency*, synthesized from multiple player modelling studies. These traits capture common gameplay preferences and strategies, making them well-suited for driving diverse in-game behaviours.

Personality traits are represented as configurable text prompts injected into the Planner, rather than as hard-coded behaviours. This allows users to select predefined personalities or define new ones by modifying prompts, enabling behavioural diversity without changing the planning logic.

To support deployment across game environments, MIMIC-Py defines lightweight mappings from game entity types predefined by PathOS to semantically equivalent concepts in each target game. PathOS defines nine entity types in total. For example, *Enemy Hazard*, described as "A hostile character, etc., which could incite combat", can be mapped to "enemies" in some games, and to "mobs" in others. These mappings enable personality prompts to be reused across games with minimal configuration changes. The full list of entities can be found on our project website [3].

2.1.2 ***Hybrid Planning.*** Many existing LLM-based agents generate only the next immediate action from the current game state [20, 23, 28], a strategy we refer to as *Bottom-Up Planning*. While effective for short-horizon or exploratory tasks, this approach often struggles with complex testing objectives that require long-horizon reasoning and sustained goal tracking [29]. For example, when tasked with crafting an in-game tool, an agent may successfully collect the required resources but later divert them to unrelated actions, ultimately failing to complete the original objective.

To address this limitation, MIMIC-Py employs a **Hybrid Planner** that dynamically switches between Bottom-Up and Top-Down strategies to track goals and task progress better. Bottom-Up Planning enables fine-grained, reactive interactions, while Top-Down Planning decomposes high-level objectives into ordered sub-tasks, providing explicit goal structure for long-horizon execution. By integrating both modes, MIMIC-Py supports complex testing objectives while retaining flexibility for exploratory behaviours, without requiring environment-specific planning logic.

To improve robustness, the Hybrid Planner further incorporates mechanisms for plan validation and revision that detect and correct infeasible or inconsistent plans, ensuring alignment with game rules and available interactions. Additional algorithmic details are provided in our prior work [2].

2.2 Action Summarizer

The LLM Action Summarizer evaluates the outcome of each executed plan and produces structured summaries of the interaction results, providing reusable, interpretable feedback to the Planner. These summaries, including the outcomes of planned actions and their relevant context, are stored in the Memory System and retrieved in subsequent iterations to guide future planning (detailed in Section 2.4). With these explicit summaries, MIMIC-Py can adapt its behaviour over time while remaining aligned with the selected personality trait.

2.3 Action Executor

The Action Executor bridges MIMIC-Py to the game under test by translating action plans into executable interactions. It is designed to support the diverse game control interfaces, enabling deployment to new environments with minimal engineering effort. The Executor supports two interaction mechanisms: *Plan-to-Parameters* and *Plan-to-Code*.

2.3.1 ***Plan-to-Parameters Translator.*** When a game exposes well-defined APIs that directly map to all in-game actions, the Action Executor translates high-level plans into API input parameters, enabling efficient interaction without code generation and supporting environments with mature control interfaces.

2.3.2 ***Plan-to-Code Translator.*** Some games expose only low-level APIs or SDKs that support basic actions, requiring testers to manually assemble code scripts for more complex behaviours. This is the case for *Minecraft* [5], where interaction relies on the Mineflayer API [19], offering primitive controls but limited support for advanced actions [2].

To operate in such environments, the Plan-to-Code Translator converts high-level plans into executable code snippets that invoke available APIs. These generated scripts, referred to as *Skills*, encapsulate reusable interaction logic that MIMIC-Py can invoke directly in subsequent executions. When a Skill fails to achieve its intended plan, the Action Summarizer provides feedback to guide the Translator's iterative refinement.

This iterative Skill construction design, centered on a growing Skill Library, enables MIMIC-Py to interact with games that lack comprehensive testing APIs without extensive manual scripting. Only a small set of example API usages is required as initial basic Skills, which guide the Translator to compose valid calls and serve as reusable functions for generating more advanced Skills. As a result, MIMIC-Py remains lightweight and practical to deploy in new or evolving game environments.

2.3.3 ***Custom Translators.*** Beyond the built-in translators, the Action Executor supports customization for games with unique or non-standard interaction mechanisms. A custom translator specifies how Planner outputs are interpreted and mapped to game-specific actions or execution routines, while reusing MIMIC-Py's existing communication and feedback infrastructure.

Implementing a custom translator requires only: (1) defining the Planner's expected output format via prompt configuration; (2) receiving plans through the existing socket-based interface; (3) executing the corresponding game actions using the game's APIs or SDKs; and (3) returning structured execution feedback (e.g., observations, logs, or errors) through the same channel. No changes to the Planner, Memory System, or Action Summarizer are required.

This extensibility is enabled by the Action Executor's encapsulated design, which isolates all game-facing logic behind a small set of interaction APIs.

The Translator type (Plan-to-Parameters, Plan-to-Code, or Custom) is selected via configuration file and determines how Planner outputs are interpreted and executed, as detailed in Section 3.2.

2.4 Memory System

The Memory System stores past interactions, including executed actions, environmental contexts, and execution summaries, collectively referred to as *Memories*, as well as reusable interaction code (*Skills*). These records are retrieved during planning and code synthesis to support context-aware decision-making and maintain personality-consistent behaviour across testing iterations.

To support long-running testing sessions, the Memory System selectively retrieves only the most relevant Memories and Skills at each iteration, rather than injecting the full interaction history into each LLM prompt. This design avoids input-length constraints, reduces inference imprecision, and enables effective reuse of prior experience.

MIMIC-Py adopts a Retrieval-Augmented Generation (RAG) approach [10] to achieve this. Both Memories and Skills are embedded in a vector database (ChromaDB [4]) for similarity-based retrieval during planning and execution. This mechanism reduces token overhead and improves inference robustness [8, 27].

The Memory System retrieves three types of information: (1) ***Preferred Memories*** aligned with the selected personality trait; (2)

Related Memories from similar past game states; and (3) ***Related Skills*** implementing interaction logic relevant to the current plan.

For personality alignment, each Memory is augmented with an LLM-generated preference summary describing how the action and its outcome reflect a given personality trait. For situational relevance, Memories store the game state at execution time. Skills are stored with natural-language descriptions of their functionality.

At runtime, retrieval uses cosine similarity over vector embeddings. Preferred Memories are retrieved by matching preference summaries against the active personality prompt, while related Memories are retrieved by matching the current game state against the stored one. Skills are retrieved by matching the current plan description against Skill descriptions. In all cases, only the top-k entries ($k = 5$ by default) are passed to the Planner or Action Executor, enabling efficient reuse of prior experience while keeping each decision step focused and computationally efficient.

3 Extensibility and Deployment

This section describes how users extend and deploy MIMIC-Py in practice. We focus on the concrete extension points exposed by the tool and the effort required to adapt it to new testing scenarios.

3.1 Extending Personality Profiles

To support diverse testing behaviours, MIMIC-Py represents personality profiles as textual prompts that encode personality traits. As shown in Figure 1 at ①, these prompts are provided as direct inputs to the Planner, decoupling behavioural variation from the underlying implementation. Users can extend the personality set by adding new prompts based on the original ones and selecting them at runtime.

3.2 Deploying into New Game Environments

Deploying MIMIC-Py to a new game environment centers on extending the Action Executor, which translates the Planner's outputs into concrete game interactions. Depending on the interfaces exposed by the target game, users can extend the plan-to-action translators in one of two ways (Section 2.3). Both options require only localized, game-oriented changes.

Option 1: API-Driven and Custom Interaction Translators. MIMIC-Py provides a unified, socket-based interaction mechanism for translating Planner outputs into game actions, which supports both the built-in *Plan-to-Parameters Translator* and fully custom translators. In this pathway, the Planner produces structured plans in a user-defined format and sends them to the game environment via the built-in WebSocket interface. MIMIC-Py then blocks until execution feedback is returned on the same channel.

When a game exposes well-defined action APIs, users can directly adopt the Plan-to-Parameters Translator (② in Figure 1) by adapting the Planner prompt templates so that generated plans conform to the parameter structure expected by the game's APIs. For games with non-standard or proprietary interaction mechanisms, users may instead define custom plan formats and implement corresponding *environment-side handlers* that interpret and execute these plans within the game runtime, while reusing MIMIC-Py's existing communication and feedback protocol.

Option 2: Code-Centric Interaction. For games that provide only low-level APIs or SDKs, users can enable the built-in *Plan-to-Code Translator* (③ in Figure 1). In this mode, the Translator generates executable code snippets for interaction.

To bootstrap this process, users must provide: (1) game specifications that include example code snippets and textual descriptions, and (2) a small set of initial Skills (helper functions) built on top of these APIs. These examples guide the code-generation process and enable the incremental construction of more advanced Skills.

In addition, users must implement a *code executor* (④ in Figure 1), which is a game-specific runtime component responsible for executing code generated by the Plan-to-Code Translator inside the game environment and returning execution feedback (e.g., observations, logs, and errors) to MIMIC-Py. The executor is invoked by the Action Executor at runtime and serves as the integration point for Plan-to-Code interaction in a new game.

Across both options, the interaction pathway is selected through a configuration file that specifies how Planner outputs are translated and executed. All game-specific implementation requirements are confined to the Action Executor and the corresponding *environment-side handlers*, i.e., game-integrated components responsible for receiving action plans or code, executing them within the game runtime, and returning execution feedback via the communication interface. The Planner, Memory System, and overall agent architecture remain unchanged, enabling scalable deployment across diverse game environments.

4 Conclusion and Future Work

This paper presents MIMIC-Py, a Python-based testing tool that operationalizes personality-driven LLM agents for practical game testing. Building on our prior research framework [2], MIMIC-Py exposes personality traits as configurable inputs and refactors planning, execution, and memory into modular components that support reusable and repeatable test generation.

Compared to the original JavaScript prototype, MIMIC-Py prioritizes deployability and extensibility. Its modular architecture decouples core agent logic from game-specific integration, enabling users to port the tool to new games through configuration, prompt adaptation, and lightweight interaction bridges rather than extensive re-engineering. This design directly addresses a key barrier in automated game testing: the high cost of adapting tools across heterogeneous game environments. We also demonstrated MIMIC-Py across multiple open-source games with differing interaction mechanisms, showing its ability to generate diverse behaviours under both API-driven and code-based interaction models.

Looking ahead, we plan to improve MIMIC-Py's efficiency by incorporating fine-tuned local models to reduce inference latency and cost, and by enhancing Skill generation with more advanced code synthesis techniques. Beyond games, the same personality-driven and modular design naturally extends to other interactive systems, such as UI testing and human–computer interaction, where behavioural diversity is essential for uncovering edge cases. Together, these contributions position MIMIC-Py as a practical foundation for scalable, behaviourally rich automated testing.

References

[1] Pierluigi Vito Amadori, Timothy Bradley, Ryan Spick, and Guy Moss. 2024. Robust Imitation Learning for Automated Game Testing. arXiv:2401.04572 [cs.LG] https://arxiv.org/abs/2401.04572

[2] Yifei Chen, Sarra Habchi, and Lili Wei. 2025. MIMIC: Integrating Diverse Personality Traits for Better Game Testing Using Large Language Model. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering* (Seoul, South Korea) *(ASE '25)*. Association for Computing Machinery, Seoul, South Korea, 39–51. doi:10.48550/arXiv.2510.01635

[3] Yifei Chen, Sarra Habchi, and Lili Wei. 2026. MIMIC-Py: A Tool for Personality-Driven Automated Game Testing with Large Language Models. https://mimic-persona.github.io/MIMIC-Py-Home-Page/

[4] Chroma. [n. d.]. chroma-core/chroma Open-source search and retrieval database for AI applications. https://github.com/chroma-core/chroma

[5] Mojang AB. TM Microsoft Corporation. 2025. Minecraft. https://www.minecraft.net/en-us

[6] Evan Debenham. 2025. Shattered Pixel Dungeon. https://shatteredpixel.com/

[7] Xidong Feng, Yicheng Luo, Ziyan Wang, Hongrui Tang, Mengyue Yang, Kun Shao, David Mguni, Yali Du, and Jun Wang. 2023. ChessGPT: Bridging Policy Learning and Language Modeling. arXiv:2306.09200 [cs.LG] https://arxiv.org/abs/2306.09200

[8] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. 2024. Retrieval-Augmented Generation for Large Language Models: A Survey. arXiv:2312.10997 [cs.CL] https://arxiv.org/abs/2312.10997

[9] Global Growth Insights. 2025. Game testing Service market. https://www.globalgrowthinsights.com/market-reports/game-testing-service-market-108174

[10] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2021. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. arXiv:2005.11401 [cs.CL] https://arxiv.org/abs/2005.11401

[11] Hao Li, Xue Yang, Zhaokai Wang, Xizhou Zhu, Jie Zhou, Yu Qiao, Xiaogang Wang, Hongsheng Li, Lewei Lu, and Jifeng Dai. 2024. Auto MC-Reward: Automated Dense Reward Design with Large Language Models for Minecraft. arXiv:2312.09238 [cs.AI] https://arxiv.org/abs/2312.09238

[12] Shunyu Liu, Yaoru Li, Kongcheng Zhang, Zhenyu Cui, Wenkai Fang, Yuxuan Zheng, Tongya Zheng, and Mingli Song. 2024. Odyssey: Empowering Minecraft Agents with Open-World Skills. arXiv:2407.15325 [cs.AI] https://arxiv.org/abs/2407.15325

[13] Newzoo. 2026. 2025 PC and console games industry year in review. https://newzoo.com/resources/blog/year-in-review-2025-to-date

[14] OpenAI. 2019. https://openai.com/index/openai-five-defeats-dota-2-world-champions/

[15] Zhen-Jia Pang, Ruo-Ze Liu, Zhou-Yu Meng, Yi Zhang, Yang Yu, and Tong Lu. 2019. On Reinforcement Learning for Full-length Game of StarCraft. arXiv:1809.09095 [cs.LG] https://arxiv.org/abs/1809.09095

[16] Wei Peng, Ming Liu, and Yi Mou. 2008. Do Aggressive People Play Violent Computer Games in a More Aggressive Way? Individual Difference and Idiosyncratic Game-Playing Experience. *Cyberpsychology & behavior : the impact of the Internet, multimedia and virtual reality on behavior and society* 11 (05 2008), 157–61. doi:10.1089/cpb.2007.0026

[17] Johannes Pfau, Jan David Smeddinck, and Rainer Malaka. 2017. Automated Game Testing with ICARUS: Intelligent Completion of Adventure Riddles via Unsupervised Solving. In *Extended Abstracts of the Annual Symposium on Computer-Human Interaction in Play (CHI PLAY '17 Extended Abstracts)*. Association for Computing Machinery, New York, NY, USA, 153–164. doi:10.1145/3130859.3131439

[18] Cristiano Politowski, Fabio Petrillo, and Yann-Gaël Guéhéneuc. 2021. A Survey of Video Game Testing. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*. IEEE, Madrid, Spain, 90–99. doi:10.1109/AST52587.2021.00018

[19] PrismarineJS. 2025. https://prismarinejs.github.io/mineflayer/#/

[20] Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. 2021. ALFWorld: Aligning Text and Embodied Environments for Interactive Learning. In *Proceedings of the International Conference on Learning Representations (ICLR)*. ICLR, Vienna, Austria. https://arxiv.org/abs/2010.03768

[21] Samantha Stahlke, Atiya Nova, and Pejman Mirza-Babaei. 2020. Artificial Players in the Design Process: Developing an Automated Testing Tool for Game Level and World Design. In *Proceedings of the Annual Symposium on Computer-Human Interaction in Play* (Virtual Event, Canada) *(CHI PLAY '20)*. Association for Computing Machinery, New York, NY, USA, 267–280. doi:10.1145/3410404.3414249

[22] Stelmaszczykadrian. 2023. GitHub - stelmaszczykadrian/Dungeon-Adventures. https://github.com/stelmaszczykadrian/Dungeon-Adventures

[23] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An Open-Ended Embodied Agent with Large Language Models. arXiv:2305.16291 [cs.AI] https://arxiv.org/abs/2305.16291

[24] Narnia C. Worth and Angela S. Book. 2014. Personality and behavior in a massively multiplayer online role-playing game. *Computers in Human Behavior* 38 (2014), 322–330. doi:10.1016/j.chb.2014.06.009

[25] Eray Yapağcı, Yavuz Alp Sencer Öztürk, and Eray Tüzün. 2025. Agents in the Sandbox: End-to-End Crash Bug Reproduction for Minecraft. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering* (Seoul, South Korea) *(ASE '25)*. Association for Computing Machinery, Seoul, South Korea, 3094–3106. doi:10.48550/arXiv.2503.20036

[26] Nick Yee, Nicolas Ducheneaut, Les Nelson, and Peter Likarish. 2011. Introverted elves & conscientious gnomes: the expression of personality in world of warcraft. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Vancouver, BC, Canada) *(CHI '11)*. Association for Computing Machinery, New York, NY, USA, 753–762. doi:10.1145/1978942.1979052

[27] Hao Yu, Aoran Gan, Kai Zhang, Shiwei Tong, Qi Liu, and Zhaofeng Liu. 2024. Evaluation of Retrieval-Augmented Generation: A Survey. arXiv:2405.07437 [cs.CL] https://arxiv.org/abs/2405.07437

[28] Zhonghan Zhao, Wenhao Chai, Xuan Wang, Li Boyi, Shengyu Hao, Shidong Cao, Tian Ye, and Gaoang Wang. 2024. See and Think: Embodied Agent in Virtual Environment. arXiv:2311.15209 [cs.AI] https://arxiv.org/abs/2311.15209

[29] Xizhou Zhu, Yuntao Chen, Hao Tian, Chenxin Tao, Weijie Su, Chenyu Yang, Gao Huang, Bin Li, Lewei Lu, Xiaogang Wang, Yu Qiao, Zhaoxiang Zhang, and Jifeng Dai. 2023. Ghost in the Minecraft: Generally Capable Agents for Open-World Environments via Large Language Models with Text-based Knowledge and Memory. arXiv:2305.17144 [cs.AI] https://arxiv.org/abs/2305.17144

A MIMIC-Py Configuration

This section describes the general configuration steps required to use MIMIC-Py. These steps are shared across all game environments and include initializing the agent and selecting personality traits. Once this configuration is completed, the execution workflow of MIMIC-Py remains consistent across different games.

A.1 Environment Setup

MIMIC-Py requires Python 3.12. After cloning the repository at https://github.com/Mimic-Persona/MIMIC-Py, install the required dependencies by running:

```
pip install -r requirements.txt
```

To simplify deployment and ensure reproducibility, we also provide a preconfigured VirtualBox virtual machine[1]. This VM bypasses most environment setup. Both options are fully documented in the repository README[2] and the following walkthrough.

A.2 Configuring Parameters

After cloning our repository, you will find a file named `.env.keep.this` in the root directory. This file serves as a template for creating your own `.env` configuration file for running MIMIC-Py. The fields that require user modifications are clearly marked in the template and README file and are illustrated in Listing 1.

The following parameters must be configured before running MIMIC-Py:

(1) Set `GAME_SUBJECT` to the target game environment. The predefined options are `MC`, `SPD`, and `DA`, corresponding to the three games described in Section B.
(2) Set `PERSONALITY` to the desired personality trait for MIMIC-Py. Supported options include: `achievement`, `adrenaline`, `aggression`, `caution`, `completion`, `curiosity`, and `efficiency`.

[1] Our Virtual Machine: The Google Drive
[2] Our Repo: https://github.com/Mimic-Persona/MIMIC-Py

```
#### General Settings ####
## Game settings ##
GAME_SUBJECT=MC

## Agent Personality Settings ##
# Choose from: achievement, adrenaline, aggression, caution,
# completion, curiosity, efficiency, or your custom personalities
PERSONALITY=achievement
# Any name for this agent
AGENT_NAME=achievement1

## Agent Settings ##
# How long do you want the agent to run in minutes
EXP_DURATION=125
# Whether to continue from previous memories with the same AGENT_NAME
IS_CONTINUED=true

## LLM Model Settings ##
# API Keys if needed
OPENAI_API_KEY=sk-proj-...

# Instruction Model
INSTRUCTION_MODEL_NAME=openai/gpt-4o
INSTRUCTION_MODEL_URL=

# Code Model (if needed)
# You have to involve this when running MIMIC-Py in Minecraft or
# other games require Plan-to-Code for interaction
CODE_MODEL_NAME=ollama_chat/XXX:Xb
CODE_MODEL_URL=http://localhost:11434
```

Listing 1: Example of a `.env` configuration for configuring MIMIC-Py.

(3) Set `EXP_DURATION` to specify the execution time limit (in minutes) for a testing session. The default value is 125 minutes.
(4) Set `IS_CONTINUED` to control whether the agent resumes from a previous testing session.
- If set to `true`, MIMIC-Py loads the Memories and Skills associated with the specified `AGENT_NAME` and continues execution.
- If set to `false`, MIMIC-Py starts a fresh session and **deletes** any existing data associated with the same `AGENT_NAME`.
- We recommend setting this option to `true`, even for first-time runs.

(5) Configure the instruction model by setting `INSTRUCTION_MODEL_NAME` and `INSTRUCTION_MODEL_URL` according to the inference backend in use.
   (a) **Using OpenAI GPT models:**
   - Set `OPENAI_API_KEY` to a valid OpenAI API key, obtainable from OpenAI.
   - The key typically follows the format `sk-XXX...` or `sk-proj-XXX...`.
   - For example, when using GPT-4o:

```
INSTRUCTION_MODEL_NAME=openai/gpt-4o
INSTRUCTION_MODEL_URL=
```

   (b) **Using Ollama models:**
   - Run the desired model using Ollama.
   - Expose the service via the default endpoint: `http://localhost:11434`.
   - For example:

```
INSTRUCTION_MODEL_NAME=ollama_chat/XXX:Xb
INSTRUCTION_MODEL_URL=http://localhost:11434
```

(6) When running MIMIC-Py in environments that require code-based interaction (e.g., Minecraft using Plan-to-Code), configure the code generation model by setting `CODE_MODEL_NAME` and `CODE_MODEL_URL` as shown in (5).
(7) At this stage, the general configuration of MIMIC-Py is complete. After following the game-specific configuration steps in Section B, MIMIC-Py can be deployed to the target game environment by running `run.py` under the root directory.

B Game-Specific Configuration

This section describes configuration steps specific to each game environment where MIMIC-Py is deployed. These steps are required only to enable MIMIC-Py to interact with a given game and are independent of MIMIC-Py's internal complexity. Instead, they reflect minimal environment-specific requirements imposed by each game's interaction interfaces.

B.1 Dungeon Adventures

This subsection describes the configuration required to connect MIMIC-Py to the *Dungeon Adventures* environment used in our evaluation.

Launching Dungeon Adventures.

(1) Ensure that a Java Development Kit (JDK) is installed on your machine. We tested MIMIC-Py with Java 22. Users who are using our virtual machine can skip this step.
(2) Start the game by running the main application entry point: `./src/main/java/com/codecool/dungeoncrawl/App.java`
(3) In the game window, click `New Adventure`, then select `Start the Game` to launch the game and initialize the server.
(4) Once launched, the terminal will display initialization logs. A shortened example of the output is shown below:

```
...
$$ Agent Mode Enabled!
```

Note. The following error messages may appear in the console. These messages originate from the JaCoCo code coverage tool and do *not* affect the functionality of either MIMIC-Py or the game. They can be safely ignored.

```
java.net.ConnectException: Connection refused: connect
...
```

Running MIMIC-Py in Dungeon Adventures.

(1) Once the game is running and the agent mode is enabled, start MIMIC-Py by running the `run.py` script. This can be done either by executing the script directly or by running the following command from the project directory:
```
python ../run.py
```
(2) After launching MIMIC-Py, the terminal will show initialization logs indicating that the agent has successfully connected to the game environment. A shortened example is shown below:

```
...
Agent connected to WebSocket server.
Memory collection initialized.
...
```

(3) Once MIMIC-Py connected to the game, press `B` to start the agent.

B.2 Shattered Pixel Dungeon

This subsection describes the configuration required to connect MIMIC-Py to the *Shattered Pixel Dungeon* environment used in our evaluation.

Launching Shattered Pixel Dungeon.

(1) Ensure that a Java Development Kit (JDK) is installed on your machine. We tested MIMIC-Py with Java 21. Users who are using our virtual machine can skip this step.
(2) Navigate to the `./MIMIC_Shattered_Pixel_Dungeon` directory:
```
cd ./MIMIC_Shattered_Pixel_Dungeon
```
(3) Start the game in debug mode by running:
```
./gradlew desktop:debug
```
(4) In the game window, click `Enter the Dungeon`, then select `New Game`. Choose *Warrior* as the character and click `Start` to begin the game.
- For consistency with our evaluation, only the **Warrior** character is tested in MIMIC-Py.

(5) Once launched, the terminal will display initialization logs. A shortened example of the output is shown below:

```
> Task :desktop:debug
[Controllers] added manager for application
[GAME] @@ You descend to floor 1 of the dungeon.
...
[GAME] $$ Game Server Opened!
...
```

Note. The following error messages may appear in the console. These messages originate from the JaCoCo code coverage tool and do *not* affect the functionality of either MIMIC-Py or the game. They can be safely ignored.

```
java.net.ConnectException: Connection refused: connect
...
```

Running MIMIC-Py in Shattered Pixel Dungeon.

(1) While in the game, press the "B" key. A pop-up window will appear, prompting you to enter a command. At the same time, the console should display:

```
[GAME] $$ MIMIC Mode Started with XXX milliseconds!
```

(2) Once the MIMIC mode is active, start MIMIC-Py by running the `run.py` script. This can be done either by executing the script directly or by running the following command from the `./MIMIC_Shattered_Pixel_Dungeon` directory:
```
python ../run.py
```
(3) After launching MIMIC-Py, the terminal will show initialization logs indicating that the agent has successfully connected to the game environment. A shortened example is shown below:

```
...
Agent connected to WebSocket server.
Memory collection initialized.
...
```

(4) Finally, return to the game pop-up window, enter the command 1, and click the `Set` button to start MIMIC-Py.

B.3 Minecraft

This subsection describes the configuration required to connect MIMIC-Py to the *Minecraft* environment used in our evaluation.

Minecraft Environment Setup (External). To run MIMIC-Py in Minecraft, users must set up a compatible Minecraft environment by installing the required mods and data packs and starting a local LAN server. To simplify deployment and ensure reproducibility, we also provide a preconfigured VirtualBox virtual machine[3] that bypasses manual environment setup. Both options are fully documented in the README for Minecraft[4].

Once the Minecraft world is launched and opened to LAN, the game will display a port number indicating the active local server (see Figure 2). Record this port number and set it as the value of `MC_PORT` in the `.env` file located at the root directory. This port is used by MIMIC-Py to connect to the Minecraft server. Detailed instructions for configuring the `.env` file for Minecraft-specific parameters are provided later.

If users choose to use the virtual machine we provided, they can skip the next step to the `Configuring Parameters` step.

Installing Node.js and Dependencies. MIMIC-Py interacts with Minecraft via a third-party API library [19], which requires Node.js. The following steps describe how to install Node.js and the required dependencies for enabling interaction between MIMIC-Py and the Minecraft environment.

(1) Ensure that `Node.js` is installed on your machine. **Important:** Only `Node.js LTS` versions (e.g., 22.x or 24.x) are supported. Newer Node versions (e.g., 25) are incompatible with `mineflayer` due to deprecated APIs. Node.js can be downloaded from the official website.

[3] Our Virtual Machine: The Google Drive
[4] Our README for MIMIC-Py in MC: MC README

Figure 2: Confirmation message showing the LAN server port number.

(2) Open a new terminal and navigate to the `./MIMIC_Minecraft` directory:
```
cd ./MIMIC_Minecraft
```
(3) Install the required Node.js dependencies:
```
npm install chromadb@1.10.5
npm install
```
(4) Install the modified `mineflayer-collectblock` plugin:
```
cd ./mc_env/mineflayer/mineflayer-collectblock
npm install mineflayer-collectblock
```
(5) Navigate to the directory containing the modified Mineflayer library and install its dependencies:
```
cd ./mc_env/mineflayer
npm install
```
(6) If compatibility issues arise during execution, remove all `node_modules` directories in the paths above and reinstall dependencies.
(7) If a `MODULE_NOT_FOUND` error related to mineflayer-collectblock occurs, please try to reinstall the 'mineflayer' package after making sure the 'mineflayer-collectblock' is correctly built from step 5..

```
#### Settings for Minecraft Only ####
## Minecraft Host and Port ##
MC_HOST=localhost
MC_PORT=60790

## Minecraft Task Settings ##
# Choose from:
# combat_1_cave_spider, combat_1_skeleton, combat_1_spider,
# cook_1_meat, harvest_1_diamond, harvest_1_sugar,
# sheer_1_sheep, and survive_for_1_day (or your custom tasks).
MC_TASK=shear_1_sheep
MC_TASK_ID=0 # Any number
MC_MONSTER_TYPE=cave_spider # Type of monster for combat tasks
```

Listing 2: Example of a `.env` configuration for running MIMIC-Py in Minecraft.

Configuring Parameters. An example configuration for this step can be found in the same `.env.keep.this` file located in the root directory, as mentioned earlier. In this subsection, we focus on parameters specific to the Minecraft environment. These parameters are highlighted in Listing 2.

(1) Set `MC_PORT` to the port number of the Minecraft LAN server started in the previous step.
(2) Set `MC_TASK` to specify the task that MIMIC-Py should perform in Minecraft. The following predefined tasks are supported:
- `combat_1_cave_spider`
- `combat_1_skeleton`
- `combat_1_spider`
- `cook_1_meat`
- `harvest_1_diamond`
- `harvest_1_sugar`
- `sheer_1_sheep`
- `survive_for_1_day`

Note: Tasks outside this list do not have predefined task descriptions. Users may still specify arbitrary task names, but such tasks will not be associated with structured task descriptions for the Planner.
(3) Set `MC_MONSTER_TYPE` to specify the monster type used in combat-related tasks. Tested options include:
- `cave_spider`
- `skeleton`
- `spider`

Note: This parameter is required only when a combat-related task is selected. The specified monster type will be spawned automatically after one in-game day (approximately 20 minutes) for the agent to engage. Other monster types supported by Minecraft may also be used.

Running MIMIC-Py in Minecraft.

(1) After completing all configurations in the `.env` file, start MIMIC-Py by running the `run.py` script. This can be done either by executing the script directly or by running the following command from the `./MIMIC_Minecraft` directory:
```
python ../run.py
```
(2) Once launched, the terminal will display initialization logs. A shortened example of the output is shown below:

```
[INFO] File written successfully
...
Subprocess mineflayer started with PID XXX.
mineflayer ready line: Server started on port 3000
...
INFO:Socket: Connected to localhost
INFO:mineflayer: Python bridge connected
...
Memory System initialized
Skill System initialized
```

(3) At the same time, status messages confirming that MIMIC-Py has successfully connected will appear in the Minecraft chat window, as illustrated in Figure 3.
(4) Press the "T" key to open the in-game chat window, then send message "b" to start the MIMIC-Py agent.

Figure 3: Minecraft chat window showing successful connection to MIMIC-Py.

C Deploying MIMIC-Py to New Game Environments

This section outlines the minimal steps required to deploy MIMIC-Py to a previously unsupported game environment. The design intentionally isolates game-specific engineering effort from the core agent logic, allowing new environments to be integrated without modifying the Planner, Memory System, or Skill System.

C.1 Game State Representation

To support decision making, MIMIC-Py requires a structured representation of the current game state.

- Users must define a game-state abstraction that captures all information relevant to planning and execution, such as player status, environment context, inventory, and nearby entities.
- This representation should be extractable from the game environment at runtime and serializable for communication with the agent.

Once defined, users implement environment-side logic to extract and transmit the game state whenever the agent requests it.

C.2 Prompt Adaptation (Repository-Guided)

MIMIC-Py relies on prompt templates to ground planning and reasoning in game-specific mechanics. Rather than detailing prompt engineering in this paper, we abstract this step as follows:

- Prompt templates are organized in the repository under `./prompts/template/`[5].
- To deploy MIMIC-Py to a new game, users copy the provided templates into a new game-specific folder and update placeholders (e.g., game state descriptions, task semantics, and game rules).
- After modifying the templates for the target game, the agent can be executed without further changes to the planning pipeline.

Detailed prompt adaptation guidelines and examples are provided in the project repository.

[5]Our prompt templates: The Templates

C.3 Game Interaction APIs

MIMIC-Py interacts with game environments via a lightweight socket-based bridge that defines a small set of interaction APIs that constitute the contract between MIMIC-Py and a game environment. By default, communication is established over a socket listening on `localhost:1111`, which can be reconfigured via the `.env` file.

Environment Interaction APIs. MIMIC-Py communicates with game environments through a socket-based bridge that exposes four core APIs. Each API has a well-defined role and invocation context during execution.

(1) `get_command()`: Blocks until a socket message with `msgType="command"` is received from the game environment, then returns the corresponding command string. This API is invoked *once at startup*, and MIMIC-Py blocks on this call until the environment sends the start signal "`b`", synchronizing agent initialization with the environment.
(2) `get_status()`: Sends a socket message with "`GetStatus`" and blocks until a response with `msgType="status"` is received. The returned value is a serialized snapshot of the current game state. This API is invoked at the *beginning of each interaction iteration* and *after each action execution* to obtain updated state information.
(3) `act_and_feedback(plan)`: Sends an action plan to the game environment using a socket message prefixed with "`ACTION:`", where `plan` is the Planner output serialized as JSON. The call blocks until the environment completes execution and responds with a message of `msgType="feedback"`, which contains (a) `logs`: execution logs generated by the environment, (b) `errors`: error messages encountered during execution. The API returns structured feedback objects derived from these fields. This API is used *only* when the Plan-to-Parameters interaction mechanism is enabled, where the environment interprets and executes structured action plans.
(4) `run_and_feedback(code, programs, timeout, executor)`: Executes generated code within the game environment using user-provided, game-specific execution support. The generated code runs with access to `programs` as helper functions and under an enforced runtime `timeout`. Execution is delegated to an `executor` function supplied by the user, which is responsible for executing the code inside the target environment and returning (i) a serialized game-state observation and (ii) execution metadata, including timeout status, log messages, and error messages. The call blocks until execution completes and returns both the observation and the execution metadata.
This API is invoked *only* when the *Plan-to-Code* interaction mechanism is enabled. A reference executor implementation is provided for Minecraft. For other game environments, users must implement a compatible executor function and register it via configuration in `run.py`.

Implementation Requirements for New Game Deployment. When deploying MIMIC-Py to a new game, users must implement the corresponding environment-side interfaces for the selected interaction mode. MIMIC-Py supports two mutually exclusive interaction modes for connecting the agent to a game environment:

- **Plan-to-Parameters:** The agent generates structured action plans (e.g., JSON), which the game environment interprets and executes via `act_and_feedback`. In this mode, users must implement environment-side handlers that map these plans to concrete game actions.
- **Plan-to-Code:** The agent generates executable code snippets that interact directly with the game environment through helper functions and are executed via `run_and_feedback`. Unlike Plan-to-Parameters, this mode requires users to implement a *game-specific code execution layer*, exposed as an `executor` function, which safely executes generated code inside the target environment and returns structured execution feedback (e.g., timeout status, logs, and errors) to MIMIC-Py. A reference implementation of such an executor is provided in `MineEnv.py`[6].
  In addition, users must supply a small set of initial *Basic Skills*, consisting of example code snippets and accompanying textual descriptions that demonstrate how the game's APIs can be invoked. These Basic Skills serve two purposes: (i) they guide the Plan-to-Code Translator in synthesizing valid code, and (ii) they act as reusable helper functions for incrementally constructing more complex Skills. Representative examples are available in the repository[7].

C.4 Registering New Environments

After implementing the required game interaction APIs, register the new environment by setting `GAME_SUBJECT` to the corresponding identifier and configuring the `IS_PLAN_TO_CODE` flag in the `.env` file to select the appropriate interaction mode.

In summary, deploying MIMIC-Py to a new game environment involves three main steps: (1) defining an abstract game state representation, (2) adapting prompt templates to reflect game-specific entities and rules, and (3) implementing a lightweight interaction bridge between MIMIC-Py and the game. These steps localize all environment-specific engineering effort, allowing the core agent architecture to remain unchanged across different testing environments.

Received 22 January 2026

[6]Example of code executor for game interaction: MineEnv.py
[7]Examples of basic Skills: The Basic Skills for MC