A Memory-Driven Action Selection Framework for Scalable Ambient NPC Behavior

1st Eric Buitron-Lopez
Department of Computer Science
The University of Western Ontario
London, Canada
ebuitron@uwo.ca

2nd Roberto Solis-Oba
Department of Computer Science
The University of Western Ontario
London, Canada
solis@csd.uwo.ca

***Abstract*—Ambient non-player characters (NPCs) in large open-world games must exhibit varied and context-appropriate behavior across large populations while operating under tight runtime budgets. Existing approaches face a fundamental trade-off: techniques such as planning and utility-based systems can produce diverse behavior but often incur significant computational cost, whereas lightweight reactive methods such as finite state machines and behavior trees scale well but frequently produce repetitive patterns.**

We present a scalable framework for ambient NPC behavior that promotes behavioral variety at low runtime cost. NPC behaviors are specified as directed graphs of actions that define action progressions. A bounded memory mechanism records recent decisions and favors untried or least-recently used transitions, discouraging repetitive behavior without requiring online planning, search, or adjustment of utility functions. This memory-driven action selection enables identically defined NPCs to exhibit varied behavior while maintaining the efficiency needed to manage large NPC populations.

The framework is implemented as an engine-agnostic C++ dynamic-link library (DLL) with JSON-defined behaviors, separating decision-making logic from engine specific execution systems. This design allows the same compiled library to be integrated across multiple game engines. We evaluated our approach in two demonstration scenarios implemented in Unity and Unreal Engine using the same framework. The results show observable behavioral variety across NPCs sharing the same definitions, cross-engine portability and sub-linear scaling from 50 to 200 NPCs while consuming less than 3.5% of the frame budget. These results indicate that memory-driven action selection provides a practical and reusable approach for controlling large populations of ambient NPCs in modern game environments.

***Index Terms*—non-player characters, game AI, behavioral variety, action selection, engine-agnostic, ambient behavior**

I. Introduction

Open-world games rely on large populations of ambient non-player characters (NPCs) whose behavioral believability directly affects a player's sense of immersion: vendors tending shops, guards patrolling streets, and pedestrians engaged in daily routines collectively form a dynamic background that contributes to the perception of a living game world. As these worlds grow in scale, NPC populations increase to hundreds or even thousands of characters each of which must exhibit varied and context-appropriate behavior. Commercial titles illustrate both the ambition and the difficulty of this challenge: *The Elder Scrolls V: Skyrim*'s Radiant AI system combines scheduled behaviors with dynamic goal selection to give each NPC a daily routine [1], *Assassin's Creed Unity* renders crowds of up to 10,000 characters but limits full AI processing to approximately 40 agents at a time [2], and *Watch Dogs: Legion* procedurally generates unique schedules for every NPC in the city but still produces noticeable repetition in their moment-to-moment actions [3]. These examples highlight a central challenge in ambient NPC control: maintaining behavioral variety across large populations under strict computational constraints.

Existing approaches to NPC behavior fall within the broad categories of deliberative and reactive methods. Planning-based algorithms such as Goal-Oriented Action Planning (GOAP) [4]–[6], Hierarchical Task Networks (HTN) [7], and utility-based systems [8], [9] can generate diverse and context-sensitive behaviors but often incur significant computational cost or need substantial manual parameter adjustment, which makes them impractical for large ambient NPC populations. In contrast, lightweight reactive approaches such as finite state machines (FSMs) and basic behavior trees [9], [10] scale well to large numbers of NPCs but frequently produce predictable or repetitive behavior patterns that players can notice. Balancing behavioral variety with computational scalability remains a central challenge in the design of ambient NPC systems, motivating the development of lightweight decision mechanisms that promote behavioral variety while maintaining the efficiency needed for large NPC populations.

In this paper we introduce a scalable framework for ambient NPC behavior that addresses this challenge through memory-driven action selection. NPC behaviors are specified as directed graphs that define action progressions. Rather than computing optimal actions through planning or evaluating choices through utility scoring, each NPC maintains a bounded memory of recent decisions. The framework uses these memory records to guide future choices of NPCs, preferring untried or least-recently used behaviors. This mechanism produces behavioral variety while keeping computational complexity low, enabling large populations of NPCs sharing the same behavior definitions to exhibit behavioral diversity.

The proposed framework is designed to be engine-agnostic

* 979-8-3315-9476-3/26/$31.00 ©2026 IEEE

and data-driven. Behavioral logic is defined using JSON specifications and executed through a C++ dynamic-link library (DLL) that separates decision-making logic from engine-specific systems. This architecture allows our framework to be integrated across multiple game engines. The framework was evaluated through two demonstration scenarios implemented in Unity and Unreal Engine using the same behavioral definitions. The experiments tested behavioral variety among NPCs with identical definitions and scalability as the number of NPCs increases. Results show that our framework produces diverse behavior patterns and low computational cost, scaling from 50 to 200 NPCs while using less than 3.5% of the frame budget.

The contributions of this paper are as follows.

1) A memory-driven mechanism for ambient NPC behavior selection that operates over directed graph representations of action sequences, producing behavioral variety among large populations of NPCs.
2) An engine-agnostic implementation that separates behavioral decision-making logic from engine-specific systems, and demonstrates cross-engine portability.
3) An empirical evaluation showing behavioral diversity and that runtime grows sub-linearly as NPC population size increases.

The framework's source code, demonstration videos, and supplementary materials are publicly available at https://www.csd.uwo.ca/%7Eebuitron/.

II. Related Work

A. Behavioral Decision-Making for NPCs

NPC behavior systems span a range of approaches from reactive to deliberative architecture [9]. At the reactive end, FSMs and behavior trees are widely used in commercial games because they offer predictable execution and low per-decision computational cost, making them suitable for large NPC populations. Although these approaches scale well, they often produce repetitive behavior patterns.

At the deliberative end of the spectrum, GOAP [4]–[6] and HTN planning [7] search for sequences of actions that satisfy character goals, producing varied and context-sensitive behavior, but incur higher per-decision computational cost. Utility systems occupy a middle ground between reactive and deliberative methods, evaluating scoring functions to select appropriate actions under current game conditions without requiring full planning, but they need careful manual parameter adjustment of evaluation functions and weights [8]. Narrative planning approaches such as the Sabre planner [11] extend deliberative reasoning with character beliefs, intentions, and theory of mind, supporting goal-directed behavior for narrative-bearing characters at a cost that grows with the depth of reasoning.

Prior work has addressed the scalability limitations of deliberative approaches. Cardon and Jacopin [12] accelerate GOAP planning using GPU parallelism, demonstrating the ability to control thousands of NPCs per frame on a GTX 1080 graphics card. Their approach improves scalability by accelerating the planning process through specialized hardware but still relies on per-decision search over action sequences and requires GPU resources that may compete with rendering in many game architectures. In contrast, we avoid planning and instead use a memory-driven selection mechanism that produces behavioral variety with lightweight computational cost.

Bulitko et al. [13] focus on behavioral variety for ambient NPCs, proposing an evolutionary approach that generates large sets of NPC behaviors and trains deep neural networks to identify the most interesting ones. Their work addresses the problem of repetitive behavior, but relies on offline evolutionary computation and learned classifiers to generate behavior policies before gameplay. In contrast, our approach produces behavioral variety at runtime without the need of evolutionary search or trained models.

Commercial games have also developed pragmatic solutions to manage AI costs at scale. *Assassin's Creed Unity* uses a layered level-of-detail (LOD) system in which only NPCs near the player receive full behavior tree processing, while distant characters use simplified state machines or animated sprites [2]. Sunshine-Hill [14] generalizes LOD selection beyond distance-based thresholds by assigning each character a criticality score derived from factors such as observability and attention, then trading detail levels across characters to maximize perceived realism within a computational budget. *Hitman Absolution* employs time-slicing to distribute AI updates across frames, preventing any single frame from exceeding its computational budget [15]. These techniques manage cost through complexity reduction or temporal distribution rather than by modifying the decision-making algorithm itself.

The proposed strategy of favoring less-recently-used options is conceptually related to exploration mechanisms in reinforcement learning, such as Upper Confidence Bound algorithms [16] and intrinsically motivated agents [17]. However, our framework targets behavioral variety in ambient NPCs without optimizing cumulative reward or learning action policies.

B. Engine-Agnostic Game AI Frameworks

Most existing game AI systems are tightly coupled to specific engines. Unity provides AI tooling alongside third-party frameworks such as CIVIL AI [18], which offers utility-based decision-making and HTN planning for ambient NPCs but is restricted to the Unity ecosystem. Unreal Engine includes a built-in behavior tree system with blackboard-based knowledge representation [19]. While these tools simplify development within a particular engine, they limit portability: NPC behaviors created for one engine cannot be easily transferred to another.

The few existing engine-agnostic frameworks for game AI primarily focus on spatial movement rather than behavioral decision-making. For example, the Menge framework [20], provides a cross-platform framework for pedestrian crowd movement with a plug-in architecture that supports multiple

navigation algorithms. While these systems address how NPCs move through space, they do not address what actions NPCs decide to perform.

We focus on a different layer of the AI stack by addressing behavioral decision-making rather than spatial navigation. Our framework achieves game engine independence through a compiled C++ DLL exposing a public C API, with NPC behaviors defined through external JSON configuration files. Any engine that can invoke C functions and implement the coordination interface described in Section III-C can integrate the framework without modifying its behavioral logic. This design enables behavior definitions to remain consistent across engines while allowing engine-specific systems to handle animations and other execution details.

Prior work addresses NPC scalability through accelerated planning, generating behaviors offline, or reducing AI complexity through techniques such as level-of-detail systems. Our work shows how a lightweight runtime mechanism can produce behavioral variety without planning, utility evaluation, or specialized hardware, enabling efficiently managing large NPC populations while avoiding repetitive behavior.

III. Framework Design

We define an entity model inspired by smart objects [21], [22], in which all interactive elements in a game are *entities*. Each entity maintains a set of properties describing its current *state* (e.g. the current capacity of a bench or the energy level of an NPC) and a set of actions that it supports. Entities play one of two roles: *environment entities* (such as market stalls or benches), which serve as interaction targets, and *behavioral entities*, which represent NPCs whose actions are selected by the framework's decision-making mechanism.

Communication between game engine systems and the framework occurs through a public interface used when game entities are created or removed. During registration an internal representation is created for each game object as either an environment entity or a behavioral entity. During unregistration all references to the entity are removed from the framework's internal decision-making structures. This separation allows the framework to focus only on behavioral decision-making while the game engine manages object lifecycles, rendering, and physical simulation.

A single mechanism, called a *state operation*, is used for evaluating conditions and modifying entity state. Each state operation specifies four components: a target indicating which entity or condition is accessed, a state key identifying which property is being evaluated or modified, an operation type defining whether the operation performs a comparison or modification, and a value used in the operation. The target can refer to the NPC itself, another entity involved in the interaction, a game environmental condition (such as weather or time of day), or the spatial distance between two entities.

A. Behavior Representation

An *action* represents a behavior an NPC can perform, such as browsing a market stall or sitting on a bench. Each action specifies a set of preconditions, which must evaluate to true before the action begins, as well as effects that modify entity state when the action starts or finishes. Actions also specify an expected duration and might define interruption behavior specifying how an NPC responds to a higher-priority event that interrupts an action. This representation allows NPC behavior to be specified through actions and state operations.

Actions fall into two categories: *entity-requiring actions*, which need another entity as an interaction target (e.g., "sit on bench" requires a bench entity), and *independently-performed actions*, which an NPC performs without a target (e.g., "wander"). For entity-requiring actions, the framework queries all registered entities that declare support for the action, evaluates each candidate's preconditions against current game conditions, and selects among valid candidates using the memory-driven selection algorithm described below.

Actions are organized into *behavior sequences*, which define the possible paths an NPC may follow when selecting its next action. Each sequence is represented as a directed graph in which nodes correspond to steps in the sequence and edges represent transitions between steps. There are three types of nodes: *action nodes*, *nested sequence nodes* (which enable complex behaviors to be composed from simpler ones), and *end nodes* (which mark the completion of a sequence). *Transitions* connect nodes and can specify preconditions. Decision points occur when a node has multiple outgoing transitions whose preconditions are satisfied. These transitions define the set of candidate behaviors among which the framework must select.

The structural components described above define the space of possible behaviors but do not by themselves produce behavioral variety. When two identically defined NPCs encounter the same decision point under identical conditions, traditional reactive methods typically produce identical behavior unless explicitly augmented with stochastic transitions or external randomization. In contrast, deliberative and utility-based systems compute numerical desirability scores or use planning algorithms to produce behavioral variation, but at high computational cost.

B. Memory System and Selection Algorithm

To address the problem mentioned above, we propose a memory-driven selection system that records the recent decisions of each NPC and uses this information to guide future action selection away from repetition. The memory-driven selection algorithm encourages exploration of alternatives by prioritizing alternatives that an NPC has not yet performed, or based on recency. This produces behavioral variety while maintaining low computational cost.

Each NPC maintains three types of memory records. *Transition memories* record which outgoing transition was selected at a decision point, storing both the destination node and the sequence in which the transition occurred. *Action memories* record which entity was selected as the target of an entity-requiring action, storing both the entity and the action. *Interruption memories* serve a different role: rather than introducing

behavioral variety, they preserve the context needed to resume an action that was interrupted by a game event. All memory records also store the timestamp at which a decision was made, enabling the framework to determine which option was used least recently.

Each type of memory record is stored in a container with a configurable maximum capacity. When the container exceeds this capacity, the oldest memory is removed, allowing the NPC to gradually forget earlier decisions.

Algorithm 1 presents the memory-driven selection algorithm used at two types of decision points: selecting which transition to follow within a sequence and selecting which entity to interact with when executing an entity-requiring action. The algorithm combines exploration and recency: it first identifies candidate options that have not yet been recorded in the NPC's memory and selects randomly among them (lines 1–13), encouraging exploration of new alternatives. If all candidates have been previously used, the algorithm examines the corresponding memory records and identifies those with the oldest timestamps (lines 14–23). Among least-recently used candidates the framework randomly selects one to prevent deterministic tie-breaking (line 24). This strategy distributes decisions over time while preventing repetitive NPC behavioral patterns.

For example, if NPC A has a transition memory for "browse vendor stall" but none for "sit on bench", the algorithm selects "sit on bench" as an unexplored alternative. If NPC B has memories for both options, the algorithm instead selects whichever option was used least recently. As a result, two NPCs with identical behavior definitions make different choices without requiring planning or utility-based scoring functions.

Our framework includes mechanisms that maintain consistent NPC behavior when unexpected conditions arise during execution. Each NPC defines *interruption sequences*, which specify behaviors to follow when an ongoing action is interrupted by a game event. When an interruption occurs, the framework stores the context of the interrupted action in an interruption memory record, begins executing the corresponding interruption sequence, and attempts to resume the original action after the interruption sequence completes. If an NPC reaches a state in which no transitions in the current sequence are valid or no suitable entities are available for the next action, the framework activates a fallback mechanism. In this case, the NPC executes a randomly selected fallback sequence defined for the NPC. In addition, a consecutive-failure safeguard halts behavioral processing if a configurable threshold of consecutive failures is reached, preventing misconfigured NPC definitions from repeatedly consuming computational resources.

Each time an NPC starts executing a new action, the framework increments a per-NPC counter called the *action token*. Completion signals from the game engine must include both the action identifier and the action token recorded when the action began. Signals whose values do not match the framework's current state are ignored, preventing delayed or stale signals from interfering with later actions. Each action also specifies a *maximum timeout duration* after which the framework automatically completes the action, ensuring that NPCs do not remain indefinitely stuck in a single behavior. Together with the interruption and fallback mechanisms described above, these safeguards maintain the reliability of the memory-driven selection system under asynchronous game engine events.

Algorithm 1 Memory-Driven Action Selection

Input: *options*: valid transition destinations or candidate entities; *memories*: memory records storing previously selected options
Output: ID of the selected option

```
1:  unusedOptions ← empty list
2:  usedMemories ← empty list
3:  for each option in options do
4:      record ← memory record in memories matching option
5:      if record does not exist then
6:          append option to unusedOptions
7:      else
8:          append record to usedMemories
9:      end if
10: end for
11: if unusedOptions is not empty then
12:     return random choice from unusedOptions
13: end if
14: oldestTime ← ∞
15: candidates ← empty list
16: for each record in usedMemories do
17:     if record.time < oldestTime then
18:         oldestTime ← record.time
19:         candidates ← {record.optionID}
20:     else if record.time = oldestTime then
21:         append record.optionID to candidates
22:     end if
23: end for
24: return random choice from candidates
```

C. Engine-Agnostic Architecture

The proposed framework is designed around three principles: (1) behavioral decision-making is separated from the game engine systems responsible for executing actions, (2) NPC behaviors are defined through external data files rather than compiled code, and (3) integration with a game engine requires only a minimal coordination interface. This design allows the framework to provide portable decision-making logic while delegating engine-specific responsibilities to the game engine.

The framework is implemented as a cross-platform C++ dynamic library that can be compiled for Windows, macOS, and Linux. It exposes a public C API that enables integration with any engine capable of calling C functions. Game engines call the DLL during each update cycle to execute the framework's

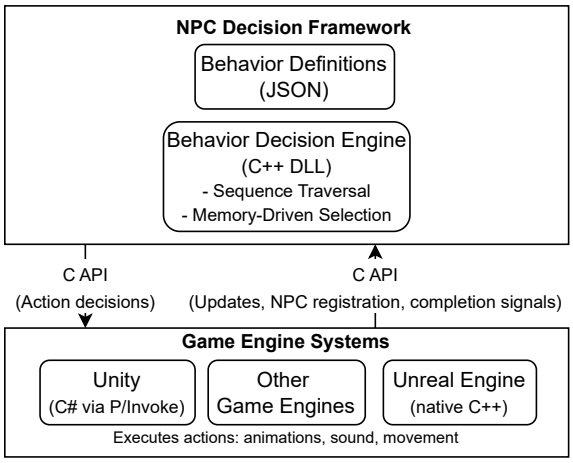

Fig. 1. Engine-agnostic framework architecture. The decision-making system is implemented as a C++ DLL exposing a C API that can be called by different game engines. NPC behaviors are defined through JSON configuration files, while engine specific systems execute the selected actions.

decision-making algorithm and receive action decisions for NPCs. The DLL performs all internal behavioral processing including sequence traversal, memory management, precondition evaluation, and action selection, while the game engine handles action execution. Fig. 1 illustrates this separation between decision-making and action execution.

Communication between the framework and a game engine occurs in both directions through a coordination interface. Outward communication occurs through callback functions supplied by the game engine when creating an instance of the framework. Inward communication occurs through a set of public C API functions that manage the framework lifecycle (creation, initialization with configuration files, and shutdown), update cycles, entity registration and unregistration, action completion signaling, and interruption handling. This interface allows the framework to be compiled once into a DLL and integrated into any engine capable of calling C functions.

Unity integrates the framework through a C# wrapper that uses P/Invoke marshalling to connect the C API, while Unreal Engine links to the DLL through a C++ wrapper layer. Other engines can integrate the framework through the provided C function interface.

To use the framework for ambient NPC decision-making, a game engine must provide three callback functions that allow the framework to interact with engine-specific systems. The first callback is invoked when the framework selects an action for an NPC and provides the engine with the NPC identifier, the current action token, the selected action, its expected duration, and the target entity if applicable. The game engine then executes the corresponding action. The second callback refreshes game environmental conditions, such as time of day or weather, when their configured update interval has elapsed. This mechanism allows the framework to incorporate dynamic conditions without direct dependencies on engine-specific systems. The third callback returns the current position of an entity, which the framework uses to evaluate distance-based preconditions during action selection.

To support engine-independent behavior definitions, the framework loads four JSON configuration files during initialization. The first file specifies the state schema, mapping human-readable property names to internal identifiers used by the framework. The second file defines game environmental conditions and their refresh frequencies. The third file defines actions including their preconditions, effects, and durations. The fourth file defines behavior sequences which specify the directed graph structure used to organize actions. Each entity also has a separate JSON configuration file loaded during entity registration, specifying its initial state, supported actions, sequence references, and memory capacities. These files describe only behavioral logic, so the same configuration files produce equivalent behavior across different game engines.

During shutdown, the framework unregisters all entities and releases allocated resources. The framework's update cycle accepts a batch-size parameter that allows game engines to distribute NPC updates across multiple calls, enabling time-sliced decision-processing. In each update cycle the framework iterates over registered NPCs, but performs decision-making only for those not currently executing an action. Entity registration and unregistration requests from the game engine are queued and processed at the beginning of the next update cycle.

IV. Evaluation

We evaluate the framework in three complementary dimensions: behavioral variety, architectural portability, and computational scalability. Behavioral variety is demonstrated through an ancient marketplace scenario, architectural portability is validated through cross-engine deployment and scalability is assessed through performance measurements across increasing NPC populations. All performance tests were conducted on a desktop PC (Windows) equipped with an AMD Ryzen 9 7900X processor, an NVIDIA GeForce RTX 2070 graphics card, and 32 GB of RAM.

A. Behavioral Demonstration

Our primary demonstration illustrates how the framework produces behavioral variety among a large number of NPCs that have identical behavior definitions. The scenario models an ancient marketplace implemented in Unity populated by 30 NPCs across three character types: 7 vendors, 5 guards, and 18 visitors. Each character type is defined through JSON configuration files specifying behavior sequences, actions, preconditions, and state properties. All NPCs of the same type share identical configuration files; any observed behavioral variation therefore emerges solely from the memory-driven selection algorithm rather than from per-character scripting.

The marketplace includes an environmental condition representing whether the market is open, which determines which behavioral paths are available to each NPC type. When the

Fig. 2. Demonstration scenarios used in the evaluation. Left: dance club in Unreal Engine. Right: ancient marketplace in Unity. Both scenarios use the same compiled DLL and behavioral logic, illustrating the framework's engine-agnostic design.

market is open, vendors sell goods from their stalls (standing or sitting depending on their energy state), visitors walk between stalls to browse merchandise, rest, or socialize when tired, and guards patrol designated areas. When the market closes, preconditions requiring an open market evaluate to false, causing NPCs to follow alternative behavioral paths: vendors leave their stalls and behave like visitors, guards continue walking but no longer perform guard duty, and visitors can no longer browse stalls and instead choose among resting, socializing, or drinking. When the market reopens, vendors receive an interruption event that triggers a "return to stall" sequence, restoring their selling routine. These context-dependent behavioral changes are implemented through preconditions and interruptions while using a single behavior definition for each character type.

The memory-driven selection algorithm produces behavioral variety at multiple levels of the sequence structure. At the top level, visitors reaching a central decision point choose among visiting a stall, resting, socializing, or drinking based on which transitions they have used least recently. Within nested sequences, additional decision points create further variation. For example, a visitor browsing a stall selects among different "browsing" actions, while a resting visitor selects among different idle behaviors. Entity selection adds another source of variation: when multiple stalls or benches satisfy an action's preconditions (e.g., available capacity and proximity to an NPC), the memory-driven algorithm selects among valid targets using action memory records. Over extended observation, NPCs sharing identical definitions follow distinct behavioral paths rather than repeating the same action patterns.

The demonstration also exercises the interruption and fallback mechanisms. A player-triggered shout event interrupts nearby NPCs triggering a "surprised reaction" sequence, after which NPCs with resumable actions return to their previous activities with preserved context (e.g., a visitor resumes browsing the same stall). When no valid target is available for the next action (e.g., all benches occupied), the fallback mechanism activates a "walk around" sequence, ensuring the NPCs remain active until conditions change.

B. Cross-Engine Validation

To evaluate the engine-agnostic design of the framework, we deployed it in both Unity and Unreal Engine. The behavioral evaluation in Section IV-A uses the ancient marketplace scenario in Unity. A second scenario, shown alongside the marketplace in Fig. 2, was implemented in Unreal Engine: a dance club environment containing 10 NPCs of two types, dancers with a looping dance sequence and visitors reusing the visitor behavior structure from the marketplace. The dance club scenario uses a simplified version of the same behavioral configuration files, adapted to its different actions and entities.

Both scenarios use the same compiled DLL without modification. The behavioral logic remains identical across both engines; only the engine-specific integration layer required adjustment to accommodate differences in coordinate systems and runtime conventions. Recorded demonstrations of both scenarios are available in the supplementary materials.

C. Performance Evaluation

The goal of this evaluation is to measure how the computational cost of the framework scales as the number of NPCs increases. Performance is measured across four NPC population sizes (50, 100, 150, and 200) using the Unity marketplace scenario. In all tests the framework iterates over all registered NPCs during each update cycle, representing a worst-case scheduling configuration.

Each configuration is evaluated in five independent runs. For each run, performance is recorded during a 60-second measurement window following a 30-second warmup period to allow the simulation to reach stable behavior. Per-frame execution time is measured using a high-resolution stopwatch wrapping the framework's update cycle. Framework cost is expressed as a percentage of total frame budget, following the evaluation methodology used by Plch et al. [23] for the *Kingdom Come: Deliverance* AI system.

Table I summarizes the performance results. As the NPC population increases from 50 to 200, the framework's average share of the frame budget increases from 1.92% to 3.41%. Quadrupling the NPC population produces only a 1.8x increase

TABLE I
FRAMEWORK TIMING PERFORMANCE. ALL TESTS PROCESS ALL NPCS EACH FRAME. EACH CELL REPORTS THE MEAN ACROSS FIVE RUNS, WITH STANDARD DEVIATION IN PARENTHESES.

NPCs	Avg FPS	Avg Framework Time (ms)	P50 Time (ms)	P99 Time (ms)	Framework Cost (% Frame Budget)
50	152.2 (1.3)	0.126 (0.002)	0.102 (0.001)	0.718 (0.016)	**1.92 (0.02)**
100	131.6 (0.9)	0.186 (0.003)	0.134 (0.002)	0.863 (0.008)	**2.45 (0.03)**
150	121.0 (0.4)	0.245 (0.003)	0.167 (0.002)	1.002 (0.012)	**2.96 (0.03)**
200	112.0 (0.5)	0.305 (0.003)	0.203 (0.002)	1.173 (0.031)	**3.41 (0.02)**

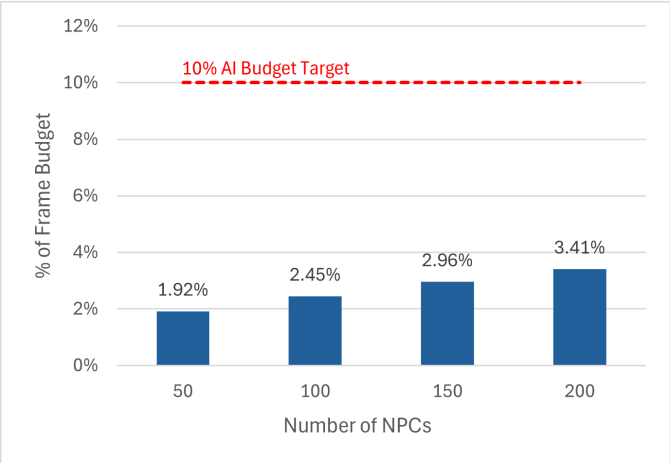

Fig. 3. Framework overhead as a percentage of frame budget across NPC populations. All configurations remain well below the 10% target threshold.

in budget consumption, showing sub-linear scaling of the decision-making workload. As seen in Fig. 3, all configurations remain well below the 10% AI budget commonly allocated in commercial games [23], [24].

This behavior can be explained by the workload characteristics of the framework. Actions have durations measured in seconds (typically 3 to 10 seconds in our tests), while the simulation runs at frame rates exceeding 100 FPS. As a result, only a small fraction of NPCs require a behavioral decision per frame. Even with 200 NPCs, the average number of NPCs requiring behavioral processing per frame is approximately 0.8 NPCs. While the framework iterates over all registered NPCs each frame, the memory-driven selection algorithm is executed only for NPCs not currently performing an action.

Additional insight is provided by the distribution of frame times. As shown in Table I, the median, P50, framework time is substantially lower than the average at all configurations (e.g., 0.203 ms vs. 0.305 ms at 200 NPCs), indicating a right-skewed distribution in which most frames require minimal processing. Occasional peaks occur when multiple NPCs complete actions simultaneously. Even at 200 NPCs, the P99 framework time remains below 1.67 ms, corresponding to 10% of the frame budget at 60 FPS.

Profiling using the Tracy profiler shows that the core behavioral decision-making operations (transition selection, entity selection, and precondition evaluation) need only $1-4$ μs per decision. The dominant per-frame cost is engine-boundary overhead such as position synchronization and action initiation callbacks that cross the DLL-engine interface. These costs are inherent to architectures that separate decision-logic from game engine systems.

For comparison, Plch et al. reported $6-12\%$ frame budget usage for the AI system in *Kingdom Come: Deliverance* [23], and commercial titles such as *Hitman Absolution* rely on time-slicing techniques to distribute AI computation across frames [15]. In contrast, our framework maintains low runtime costs without time-sliced updates.

Run-to-run variability is minimal across all configurations. The measured framework budget percentage varies by at most 0.03 percentage points across five runs, indicating consistent and reproducible performance. These results indicate that our memory-driven decision algorithms can support large NPC populations without becoming a performance bottleneck.

V. LIMITATIONS AND FUTURE WORK

The behavioral evaluation presented in Section IV-A is primarily qualitative and focuses on illustrating the observable behavior produced by the framework. While the demonstration scenarios exercise all the framework's features across multiple NPC types, they do not yet include quantitative measures of behavioral variety such as action distribution entropy or repetition frequency. Developing such metrics, together with controlled benchmarking against behavior trees, utility-based systems, and planning approaches using equivalent behavioral specifications, would enable more systematic evaluation of behavioral diversity, computational trade-offs, and responsiveness across decision-making architectures.

The empirical evaluation also has scope limitations worth noting. Performance was measured for populations up to 200 NPCs, a range consistent with typical ambient populations in commercial open-world games. Although profiling data suggest significant computational headroom for larger populations, this has not been empirically validated. Performance measurements were conducted on Windows; although the framework has been developed and tested on macOS as well, behavior and timing on other operating systems have not yet been evaluated. Cross-engine validation in Section IV-B demonstrates that the same compiled library and configuration files execute correctly in both Unity and Unreal Engine. However, the two scenarios differ in content; evaluating behavioral equivalence across engines under matched scenarios is left for future evaluation.

The memory-driven selection algorithm produces behavioral variety through the use of bounded memory but does not currently model individual NPC preferences or personality

traits. Extending the selection algorithm with weighted preferences or personality parameters could allow different NPCs to exhibit distinct behavioral tendencies while preserving computational efficiency. A related direction would dynamically adjust memory capacity or update frequency according to NPC importance or player proximity, combining memory-driven behavioral selection with level-of-detail techniques used in large-scale game AI systems.

Currently, NPC behaviors must be authored manually through JSON configuration files, requiring careful coordination among actions, sequences, and state definitions. As behavior configurations increase in complexity, maintaining consistency across multiple configuration files can become error-prone. Developing visual editors for generating and validating these configuration files would reduce authoring effort and help prevent configuration errors.

VI. Conclusion

This paper presented a memory-driven framework for scalable ambient NPC behavior that addresses the trade-off between behavioral variety and computational efficiency in large NPC populations. The central contribution is a lightweight action selection algorithm based on bounded memory in which each NPC records recent decisions and uses this information to guide future choices. By selecting among available options according to whether they have been previously explored and how recently they were used, the framework produces behavioral variety among identically-defined NPCs without requiring planning algorithms or utility-based scoring functions.

The framework's engine-agnostic architecture, implemented as a C++ DLL with JSON-defined configuration files, separates behavioral decision-making from the game engine systems responsible for executing actions. This architecture enables the same behavioral logic to be reused across multiple game engines through a minimal coordination interface. Cross-engine deployment in Unity and Unreal Engine shows that the same compiled DLL and configuration files produce consistent behavior in different development environments.

Performance evaluation shows that the framework scales efficiently with increasing NPC populations. In the evaluated scenarios, runtime overhead grows sub-linearly as the number of NPCs increases, remaining below 3.5% of the frame budget for populations up to 200 NPCs. Profiling results indicate that the dominant runtime cost arises from engine-boundary communication rather than from the behavioral decision-making algorithm, suggesting that the framework can support larger NPC populations without becoming a performance bottleneck.

These results demonstrate that simple memory-based decision algorithms can provide a practical and scalable alternative to planning or utility-based systems for controlling large populations of ambient NPCs in modern open-world games.

Acknowledgments

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) through Discovery Grant RGPIN-2026-07898.

Cette recherche a été financée par le Conseil de recherches en sciences naturelles et en génie du Canada, numéro de référence RGPIN-2026-07898.

References

[1] M. Bertz, "The Elder Scrolls V: Skyrim preview - the technology behind The Elder Scrolls V: Skyrim," Game Informer, 2011, Accessed: May 11, 2026. [Online]. Available: https://gameinformer.com/games/the_elder_scrolls_v_skyrim/b/xbox360/archive/2011/01/17/the-technology-behind-elder-scrolls-v-skyrim.aspx

[2] F. Cournoyer and A. Fortier, "Massive crowd on Assassin's Creed Unity: AI recycling," 2021, Accessed: May 11, 2026. [Online]. Available: https://www.youtube.com/watch?v=Rz2cNWVLncI

[3] C. Dragert, "Watch Dogs: Legion — the design and tech behind play as anyone," 2020, Accessed: May 11, 2026. [Online]. Available: https://www.youtube.com/watch?v=mMFqK7dn5Wo

[4] J. Orkin, "Applying goal-oriented action planning to games," AI Game Programming Wisdom 2 draft, 2003, Accessed: May 11, 2026. [Online]. Available: https://web.archive.org/web/20230912173044/https:/alumni.media.mit.edu/%7Ejorkin/GOAP_draft_AIWisdom2_2003.pdf

[5] J. Orkin, "Agent architecture considerations for real-time planning in games," *AIIDE Conference*, vol. 1, no. 1, pp. 105–110, 2005.

[6] J. Orkin, "Three states and a plan: The A.I. of F.E.A.R." in *Proceedings of the Game Developers Conference (GDC)*, 2006, Accessed: May 11, 2026. [Online]. Available: https://www.gamedevs.org/uploads/three-states-plan-ai-of-fear.pdf

[7] T. Humphreys, "Exploring HTN planners through example," in *Game AI Pro*. CRC Press, 2013, vol. 1, pp. 149–167.

[8] D. Graham, "Breathing life into your background characters," in *Game AI Pro*. CRC Press, 2013, vol. 1, pp. 451–458.

[9] I. Millington, *AI for Games*, 3rd ed. CRC Press, 2019.

[10] Y. A. Sekhavat, "Behavior trees for computer games," *International Journal on Artificial Intelligence Tools*, vol. 26, no. 02, p. 1730001, 2017.

[11] S. G. Ware and C. Siler, "Sabre: A narrative planner supporting intention and deep theory of mind," in *Proceedings of the AIIDE Conference*, vol. 17, no. 1, 2021, pp. 99–106.

[12] S. Cardon and E. Jacopin, "Binary GPU-planning for thousands of NPCs," in *2020 IEEE Conference on Games (CoG)*, Osaka, Japan, 2020, pp. 678–681.

[13] V. Bulitko, S. Carleton, D. Cormier, D. Sigurdson, and J. Simpson, "Towards positively surprising non-player characters in video games," *AIIDE Conference*, vol. 13, no. 2, pp. 34–40, 2017.

[14] B. Sunshine-Hill, "Phenomenal AI Level-of-Detail Control with the LOD Trader," in *Game AI Pro*. CRC Press, 2013, vol. 1, pp. 185–200.

[15] M. De Pascale and M. Vehkala, "Creating the AI for the living, breathing world of Hitman: Absolution," 2013, Accessed: May 11, 2026. [Online]. Available: https://gdcvault.com/play/1017802/Creating-the-AI-for-the

[16] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine Learning*, vol. 47, pp. 235–256, 2002.

[17] P.-Y. Oudeyer and F. Kaplan, "What is intrinsic motivation? a typology of computational approaches," *Frontiers in Neurorobotics*, vol. 1, p. 14, 2007.

[18] B. Tree Limited, "Introduction to CIVIL-AI-SYSTEM," 2026, Accessed: May 11, 2026. [Online]. Available: https://bardtreelimited.com/docs/guide-basic-intro-civil-ai-system

[19] Epic Games, "Artificial intelligence in Unreal Engine," Accessed: May 11, 2026. [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/artificial-intelligence-in-unreal-engine

[20] S. Curtis, A. Best, and D. Manocha, "Menge: A modular framework for simulating crowd movement," *Collective Dynamics*, vol. 1, p. A1, 2016.

[21] K. Dill, "Structural architecture - common tricks of the trade," in *Game AI Pro*. CRC Press, 2013, vol. 1, pp. 61–71.

[22] D. Brewer and R. Graham, "Knowledge is power: An overview of knowledge representation in game AI," 2020, Accessed: May 11, 2026. [Online]. Available: https://www.youtube.com/watch?v=Z6oZnDIgio4

[23] T. Plch, M. Marko, P. Ondracek, M. Cerny, J. Gemrot, and C. Brom, "An AI system for large open virtual world," *AIIDE Conference*, vol. 10, no. 1, pp. 44–51, 2014.

[24] J. Gregory, *Game Engine Architecture*, third edition ed. Boca Raton London New York: CRC Press, Taylor & Francis Group, 2019.