

Article

A Resilient and Extensible Digital Curling Platform Using Server-Sent Events and REST APIs

Ryosuke Katagiri ^{1,*} , Tadaaki Sone ^{2,3} , Shimpei Aihara ³  and Takashi Kawamura ¹ 

¹ Faculty of Textile Science and Technology, Shinshu University, 3-15-1 Tokida, Ueda 386-8567, Nagano, Japan; kawamura@shinshu-u.ac.jp

² Department of Electrical, Electronic and Information Engineering, Faculty of Engineering, Kanagawa University, 3-27-1 Rokkakubashi, Yokohama 221-8686, Kanagawa, Japan; ft102256kn@jindai.jp

³ Japan Institute of Sports Sciences, 3-15-1 Nishigaoka, Kita-ku, Tokyo 115-0056, Japan; shimpei.aihara@jpnssport.go.jp

* Correspondence: 26hs106d@shinshu-u.ac.jp

Abstract

Curling is strategically complex, yet data-driven strategic AI is limited by the scarcity of large-scale match data with shot-by-shot states and histories. Digital Curling provides such an environment, but no prior system has combined match resumption after failures, centralized state persistence, and a decoupled simulator into a single, reusable research platform. We propose and implement a new Digital Curling platform that improves resilience and extensibility through a stateless server, database-backed state persistence, REST APIs and Server-Sent Events (SSE), and a decoupled simulator, supporting both the standard four-player and mixed-doubles formats. In a 24 h experiment, the system recovered from all 290 timeout-induced communication failures with 100% reconnection success and a mean recovery time of about 1 s. In a server-restart experiment ($N = 50$), it resumed every match with full state consistency, and it ran for about 60 h over six days at Expo 2025 Osaka, Kansai, with no system-attributable failures. A load evaluation further shows that the platform sustains concurrent load and scales near-linearly, achieving $1.97\times$ throughput across two server workers. Under these controlled conditions, the platform offers a promising foundation for continuous match operation and future strategic AI research in curling.

Keywords: curling; Digital Curling; AI; communication protocols; mixed doubles; REST APIs; Server-Sent Events



Academic Editor: Andrea Prati

Received: 29 May 2026

Revised: 29 June 2026

Accepted: 6 July 2026

Published: 9 July 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

Curling is a winter sport characterized by a high degree of strategic complexity. A match consists of multiple ends, in each of which the two teams alternately deliver stones. The score for an end is determined by the number of stones belonging to one team that lie closer to the center of the house—the target area on the ice—than the opponent's nearest stone. Accordingly, success in curling depends not only on the execution of individual shots but also on tactical decision-making over multiple shots. Players must build favorable board positions while anticipating the final arrangement of stones at the conclusion of the end. In addition to the standard four-player format, curling also includes mixed doubles, in which each team consists of one male and one female. Mixed doubles differs from the four-player format in several rules, including the number of shots per end, the initial stone placement, and the rules governing which team has the last-stone advantage in each end;

consequently, its tactical structure and simulation requirements also differ. Supporting both game formats within a unified simulation platform is therefore a practical requirement for research infrastructure that aspires to cover the breadth of competitive curling, as well as for interactive deployments in which human players compete against AI opponents. In both formats, curling constitutes a strategic decision-making problem with a continuous action space and inherent uncertainty, leading to an enormous strategy space.

Despite this complexity, tactical decision-making in practice still depends heavily on the experience of players and coaches, and a systematic analysis and general formulation of effective strategies have not yet been fully established. Against this background, curling has increasingly attracted attention as a target domain for AI-based strategic analysis. Recent advances in deep learning and search algorithms have increased the feasibility of developing highly capable strategic AI; however, training such systems requires large volumes of match data covering diverse game states. Furthermore, modern strategic AI models are increasingly expected to leverage not only single board states but also shot histories and state transitions as temporal information. Because collecting large-scale, per-shot game-state data from real-world curling is impractical, it is essential to establish a research platform that can generate diverse game states and strategies computationally and record both the latest board state and the full per-shot history throughout a match. The goal of this work is this research infrastructure itself: a resilient, extensible platform on which AI agents can play and from which consistent, large-scale match data can be collected.

Among existing computer-based simulation environments for curling, Digital Curling has been the most widely adopted as a competitive simulation platform for AI-versus-AI matches [1–4]. However, when viewed as a research infrastructure for modern data-driven strategic AI, the conventional system has several limitations, particularly in terms of recoverability, data accessibility, and extensibility: (1) inability to resume matches upon network disconnection or server failure, (2) inadequate data-sharing infrastructure due to decentralized log management, and (3) difficulty in standalone use and extensibility caused by tight coupling of the simulator. First, because communication relies on persistent connections and the game state is maintained in server memory, temporary communication failures or server crashes prevent ongoing matches from continuing and can cause the loss of intermediate computation results and match-progress data. This problem significantly reduces data-collection efficiency and experimental reproducibility, particularly in matches between AI agents that require long computation times. Second, match logs are stored separately on the local storage of individual servers, and no infrastructure has been established for centrally managing, sharing, and reusing historical match data across researchers. As a result, not only is it difficult to accumulate large-scale training data, but reliance on logs generated against only a limited set of opponents can also bias the distribution of board states and adopted strategies, thereby hindering the training of strategically diverse AI agents. Third, the simulator is embedded within the server program. This design makes it difficult to invoke the simulator directly from AI-development environments such as Python or to run simulations in parallel as independent processes. Moreover, such tight coupling reduces flexibility in adapting to future rule revisions, adding new functionality, and maintaining the system. A detailed comparison between the conventional system and the system proposed in this study is provided in Section 2.2 (Table 1).

Table 1. Comparison of functional characteristics between the conventional system and the proposed system, together with the design significance of each change. In this paper, the conventional system refers to the third-generation Digital Curling system.

Comparison Aspect	Conventional System	Proposed System	Design Significance (Why Non-Trivial)
Communication Architecture	Persistent socket communication (Dedicated protocol handling required)	HTTP-based communication using REST APIs and SSE (Easier interoperability and reconnection handling)	Choosing REST + SSE over WebSocket/gRPC yields browser-native reconnection without any application-layer connection management.
State Retention	In-memory dependency (Match state is lost when the server process stops)	Database persistence + stateless server (Recoverable after server restart)	All match state is externalized to a single source of truth at a granularity that permits exact mid-match resumption.
Resilience (Recovery from disconnection)	Disconnection leads to match interruption (No reconnection-based recovery mechanism)	Automatic reconnection and state resynchronization (Continuation from the latest persisted state)	The current end's shot history is resynchronized on reconnect so that sequence-model inputs remain continuous.
Client Environment Compatibility	Dedicated implementation required (Strong dependency on proprietary socket protocol)	Web-standard-based access (Usable from multiple languages, OSs, and browser environments)	Eliminating any proprietary protocol driver lets agents in any language or runtime connect over standard HTTP.
Horizontal Scalability	Difficult (Persistent connections and process-local state hinder scaling)	Facilitated (Stateless design + Redis Pub/Sub event delivery)	Decoupling per-match state from the process lets multiple stateless servers share it and fan out updates via Redis Pub/Sub.
Data Sharing & Reuse	Difficult (Logs are locally stored as per-shot files and not centrally retrievable)	Facilitated (Centralized RDB storage + standardized retrieval through API)	Per-shot states are persisted in a queryable schema designed for sequence-model dataset construction, not merely logged to disk.
Simulator Accessibility	Embedded in the server program (Direct standalone use is difficult)	Decoupled as an independent library (Direct use from Python and standalone large-scale simulation are possible)	A pybind11 boundary lets the same physics engine run both in-server and standalone for large-scale data generation.
Supported Game Modes	Standard four-player mode only	Standard four-player mode + mixed doubles mode	Both formats are mapped onto one fixed 16-slot board representation with format-specific rule selection.
Extensibility	Tightly coupled (Physics, rules, and communication are strongly intertwined)	Loosely coupled (Independent simulator and modular server design)	Separating physics, rules, and transport allows each concern to evolve independently.

In this study, we therefore propose and implement a new Digital Curling platform with improved resilience and extensibility. We position this work as a systems and research-infrastructure contribution rather than a new algorithm or communication technology, while REST APIs, SSE, relational databases, and containerization are individually well-established, no prior Digital Curling system has combined, in a single platform, match resumption after failures, centralized state persistence, a Python-accessible decoupled simulator, and unified four-player/mixed-doubles support. The contribution of this work therefore lies in identifying the specific failure modes of Digital Curling infrastructure, mapping each to a deliberate architectural choice (Section 3), and demonstrating empirically that the resulting integration is sufficient for stable AI-research operation and for a multi-day real-world deployment. Our basic design principle is to reformulate the limitations of the conventional system into two requirements: resilience, which enables match continuation and state recovery from communication failures and server restarts, and extensibility/portability, which supports diverse client environments and operational modes while accommodating future functional and operational extensions. The main contributions of this study are as follows.

- By designing the server to be stateless and persisting all match states in a database, the system enables state restoration and match continuation even in the event of communication interruptions or server failures.
- By adopting an HTTP-based communication architecture that combines REST APIs (client-to-server) with SSE (server-to-client), the system obtains automatic, browser-native reconnection without application-layer reconnection logic and becomes usable from a wide range of execution environments, including standard web browsers and multiple programming languages, without custom protocol drivers.
- By separating the simulator as an independent library and enabling direct use from Python environments as well as parallel execution of simulation tasks, the system provides greater flexibility for AI development and large-scale data generation. In addition, by combining an event-driven structure with containerized operations, the platform is designed to accommodate synchronization across multiple server processes and future horizontal scaling.

The versatility of this design was further demonstrated at Expo 2025 Osaka, Kansai, where the platform served as the backend for a six-day interactive curling exhibition in which general visitors competed against an AI opponent—a use case orthogonal to the primary AI research scenario—without requiring any architectural modifications.

The remainder of this paper is organized as follows. Section 2 reviews related work on curling AI, existing Digital Curling systems, and candidate web communication protocols and identifies the gaps that motivate the proposed platform. Section 3 describes the design requirements and the implementation of the proposed platform. Section 4 presents the resilience experiments and the real-world deployment results, together with discussion. Section 5 concludes the paper and outlines future work.

2. Related Work

This section reviews prior work along three lines that situate the contributions of this paper: (i) AI methods for curling strategy and decision-making; (ii) existing Digital Curling systems and their architectural limitations; (iii) candidate web communication protocols for real-time client–server interaction.

2.1. Curling AI and Strategic Decision-Making

Research on AI for curling has primarily focused on coping with the sport’s continuous action space and the inherent uncertainty of stone trajectories. Katoh et al. proposed a curling AI that explicitly incorporates the probabilistic nature of shot outcomes into the strategy-evaluation process, thereby selecting shots whose expected value is robust to execution noise [5]. They subsequently extended this line of research with a game-tree search method that propagates uncertainty through the search, enabling more reliable evaluation of multi-shot tactics [6]. Iwasaki et al. studied the learning of evaluation functions and the prediction of winning probability based on score difference and the number of remaining ends [7], providing a basis for end-level strategic judgment. Ohto and Tanaka applied Monte Carlo Tree Search to curling AI and demonstrated its effectiveness for shot-level decision-making [8]. More recently, Han et al. introduced a strategy model based on Neural Fictitious Self-Play (NFSP), aiming to learn approximate equilibrium strategies in the curling game from large amounts of self-play data [9].

Across these studies, two practical requirements consistently emerge. First, the methods rely on the availability of substantial volumes of match data that include shot-by-shot game states and shot conditions; this is especially true for learning-based methods such as NFSP and modern sequence models. Second, robust evaluation of these methods requires a stable, reproducible competition environment in which AI agents can play against

one another over extended periods. These requirements are difficult to satisfy with manually collected real-match data alone, and they motivate the use of a digital simulation platform as the data-generation and evaluation infrastructure for curling AI research.

2.2. Digital Curling Systems and Their Limitations

Digital Curling was originally proposed by Kitasei et al. as a server-based competition environment for curling AI, with the goal of providing a common platform on which AI agents from different developers can compete [1,2]. Subsequent work by Mori et al. improved the simulator to be more robust to differences in environmental conditions across executions [3]. Uehara further developed a third-generation Digital Curling system, in which the physics model was reorganized as the Friction Curl Velocity (FCV) model, and the system was made more extensible with respect to additional simulation models [4,10]. The current Digital Curling system, illustrated in Figure 1, adopts a server–client architecture in which the server integrates the curling rules and the simulator, and clients exchange shot information with the server via socket-based communication.

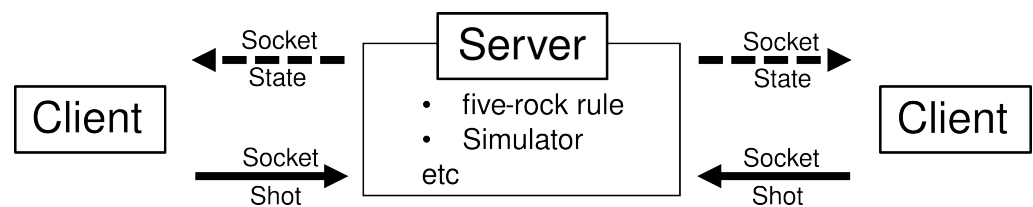


Figure 1. Overview of the conventional (third-generation) Digital Curling system. The server integrates the curling rules and the simulator. Communication between the server and the clients is performed via sockets: the server sends board-state information to the clients, and the clients send shot information back to the server.

While these prior systems have been successful in enabling AI-versus-AI competitions, they share three architectural limitations that constrain their use as a modern research infrastructure, as outlined in Section 1: limited recoverability under communication or server failures, decentralized log management that hampers data sharing and reuse across researchers, and tight coupling between the server and the simulator that limits standalone simulator use and future extensibility. Table 1 summarizes these limitations together with the corresponding design choices of the platform proposed in this paper, and Sections 3 and 4 describe the design and its evaluation in detail.

2.3. Web Communication Protocols and Real-Time Notification

Replacing the proprietary socket-based communication of the conventional system with web-standard protocols requires careful selection of the underlying communication technology. Recent surveys of internet and web-application communication protocols compare gRPC, WebSocket, Server-Sent Events (SSE), and REST APIs along several axes, including communication direction (unidirectional vs. bidirectional), browser compatibility, reconnection support, and implementation overhead [11,12]. A summary comparison of these protocols is shown in Table 2. These comparisons indicate that gRPC, while highly efficient, has well-known limitations in browser environments; WebSocket offers low-latency bidirectional persistent connections at the cost of placing connection management and reconnection handling on the application layer; SSE provides server-to-client push over HTTP with native browser support and built-in reconnection; and REST APIs offer simple, stateless request–response semantics with broad interoperability.

Table 2. Qualitative comparison of candidate communication protocols.

Protocol	Browser Compatibility	Server Push	Bidirectional	Reconnection Support	Implementation Burden
gRPC	Limited	Yes	Yes	Application-managed	High
WebSocket	Excellent	Yes	Yes	Application-managed	Medium
SSE	Excellent	Yes	No	Built-in	Low
REST API	Excellent	No	No	Stateless	Low

The communication patterns required by a Digital Curling platform are inherently asymmetric: the server pushes board-state updates and turn transitions to clients, while clients submit discrete shot requests. This asymmetry suggests a combined design that uses different protocols for the two directions, rather than a single bidirectional channel; we exploit this observation in Section 3.

In addition, when the platform is deployed across multiple server processes for fault isolation or horizontal scaling, identical update notifications must be delivered to clients regardless of which server process they are connected to. A common approach to this problem is to introduce a message broker that decouples event publication from event delivery; in particular, Redis provides a Pub/Sub mechanism in which publishers send messages to named channels and subscribers receive all messages on the channels they have subscribed to [13]. We adopt this pattern in our platform to maintain consistent state notifications under multi-server configurations.

3. Materials and Methods

This section describes the design and implementation of the proposed Digital Curling platform, starting from the design requirements and then covering the system architecture, state persistence, communication mechanism, server and client implementation, simulator decoupling, and operational infrastructure.

3.1. Design Requirements

Following standard software-engineering practice, we organize the requirements that a Digital Curling research platform must meet into *functional requirements* (services the platform must provide) and *non-functional requirements* (quality attributes the platform must exhibit). Two quality attributes are central to this study: resilience and extensibility/portability. Each requirement below is annotated with the section that evaluates or realizes it so that requirements, design choices, and evaluation are explicitly linked.

Functional requirements. The platform must:

- progress a curling match between two competing clients, invoking the simulator and updating the board, turn, and score at each shot (Section 3.5);
- support both the standard four-player format and mixed doubles within a single platform (Section 3.6);
- persist every per-shot board state and shot history so that completed and in-progress matches can be retrieved and reused for AI training (Section 3.3).

Non-functional requirements. The platform must satisfy the following quality attributes.

Resilience refers to the ability to recover and continue a match after communication or server-side disruptions:

- autonomous reconnection after a temporary communication disruption (evaluated in Section 4.1);
- service recovery following a server-process restart (evaluated in Section 4);

- restoration of the board state and match-progress information immediately before the failure (evaluated in Sections 4.1 and 4).

Extensibility/portability refers to the ability of the platform to remain usable for research across diverse execution environments and future extensions:

- the client and simulator should not depend heavily on any specific operating system or implementation language;
- the platform should be usable from a variety of environments, including web browsers;
- the architecture should be extensible to future parallel operation and event synchronization across multiple server processes (provided architecturally and validated by a two-worker scaling test; see Section 4.4).

To make the resilience requirements operationally concrete, we specify the following performance targets that guided implementation decisions: automatic reconnection should complete within 2 s under nominal network conditions; state resynchronization (board state and shot history for the current end) should complete within 100 ms of connection re-establishment, excluding intentional client-side wait times; and the system should sustain operation over multiple consecutive days of continuous use without manual intervention. Concurrent client load and multi-server scaling are evaluated in Section 4.4.

Table 3 summarizes how each design requirement maps to the specific technology choices made in this study.

Table 3. Mapping from design requirements to adopted technology choices.

Requirement	Technology/Design Choice	Rationale
Reconnection after communication disruption	SSE (built-in reconnect) + REST API (stateless request)	SSE provides automatic reconnect without application-layer logic; REST ensures each request is self-contained
State restoration after server restart	Stateless server + PostgreSQL persistence	Decoupling server-process lifetime from match state allows a fresh process to fully reconstruct any match from the database
Portability across OS and language environments	HTTP-based protocol (REST + SSE)	HTTP is universally supported across platforms, languages, and browser environments without custom protocol drivers
Multi-format support (four-player + mixed doubles)	Decoupled simulator library (C++/pybind11) + external configuration	Format-specific rules are applied as pre/post-processing layers; switching formats requires only configuration changes
Direct simulator access for AI development	Simulator as standalone Python-accessible library (pybind11)	Enables parallel standalone simulation without a running server, supporting large-scale data generation and model training
Multi-process event consistency	Redis Pub/Sub message broker	Decouples event publication from delivery, ensuring all server processes push identical notifications to their respective clients
Reproducible deployment	Docker + Docker Compose containerization	Encapsulates all runtime dependencies; identical service configurations can be deployed across development and production environments

3.2. Overall System Architecture

The proposed system comprises a server, clients, a relational database, an independent simulator, and a message broker. Figure 2 shows an overview of the proposed system.

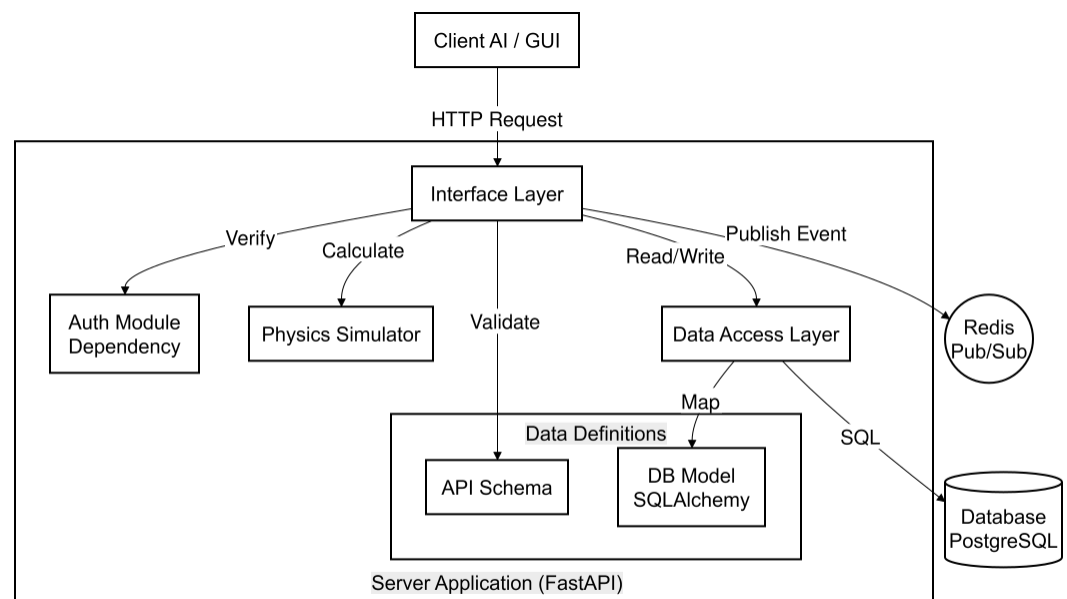


Figure 2. Overview of the proposed Digital Curling system. The server receives shot requests from clients, invokes the simulator, persists match states in the database, and distributes state updates via Redis Pub/Sub.

The server is responsible for match progression control, API provision, authentication, and state transition management. Clients receive board information from the server, determine the next shot, and send the shot information. Here, a client corresponds to one competing side (a team controlled by a single AI agent or by a human-operated application), not to an individual human player. In the four-player format, a single client is responsible for all of its team's deliveries, and likewise in the two-person mixed-doubles format; a match therefore always involves exactly two competing player clients (`team0` and `team1`), regardless of the on-ice player count, alongside any number of read-only viewer clients that may connect to observe the match. The database persistently stores match settings, board states, shot histories, and related data. The simulator is implemented as a library independent of the server and performs physical simulations based on shot information. The message broker propagates events between server processes and helps maintain consistent real-time notifications in distributed deployments. By designing these components to be loosely coupled, the system enables state restoration in the event of failures, future system extension, and large-scale operation.

3.3. State Persistence and Database Design

The server is designed as a stateless component, and data related to match progression are persistently stored in the database. The recovery mechanism based on state persistence is illustrated in Figure 3. With this architecture, even in the event of a network interruption or server restart, a match can be seamlessly resumed by restoring the latest state retrieved from the database.

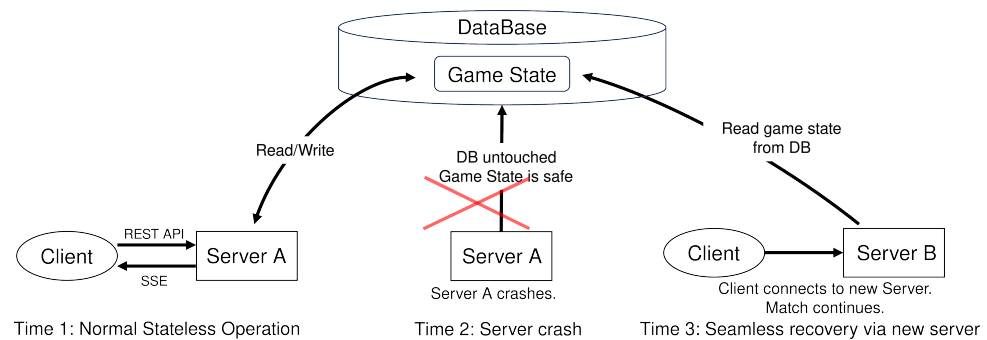


Figure 3. Server-side failure recovery mechanism: when the active server process becomes unavailable, a new server instance can resume the match by loading the latest persisted state from the database, because the server holds no match state in memory. The cross mark indicates the failure case in which the crashed server process can no longer write data to the database, so recovery proceeds from the latest state already persisted in the database.

PostgreSQL is selected as the database management system. A relational database is employed in this system to consistently manage data pertaining to match progression, history, configuration, and physics simulation. UUIDs are adopted as primary keys to prevent ID collisions in distributed environments. Furthermore, for data with complex structures and large sizes, such as stone coordinates and trajectories, the JSONB datatype is utilized to achieve both data flexibility and searchability. The data schema is organized into three layers: match management and configuration, game-state tracking and management, and shot history management. The roles of the primary tables are summarized in Table 4. In the match management layer, match settings, participating players, tournament information, and the simulator type are managed. In the game-state tracking and management layer, end numbers, shot numbers, the next player to throw, scores, and stone coordinates at each point in time are recorded. In the shot history layer, the input parameters for each shot along with references to the pre- and post-shot states are stored. By storing state and history separately in this way, match restoration is made possible while also facilitating subsequent log analysis and the construction of datasets for AI training.

Table 4. Roles of the main tables.

Table	Role
match_data	Stores the basic metadata of a match, including tournament, simulator, participating players, and final score.
state	Represents the game state at each step of the match, including end number, shot number, board coordinates, and intermediate score.
shot_info	Stores detailed information for each shot, such as the thrower, shot parameters, and the state transition before and after the shot.
player	Stores player profiles and ability parameters such as name, team affiliation, and shot accuracy characteristics.
trajectory	Stores time-series trajectory data of each shot.
stone_coordinate	Stores the stone positions on the sheet at a given state.
score	Stores score arrays for both teams.
tournament	Stores tournament metadata.
physical_simulator	Stores metadata of the physics simulator used in the match.

The complete entity–relationship diagram of the database schema is shown in Figure 4.



Figure 4. Entity–relationship diagram of the database schema. Match metadata, per-shot states, shot history, and recorded trajectories are linked by foreign keys, and mixed-doubles-specific configuration is isolated in the `match_mixed_doubles_settings` table.

3.4. Communication Architecture and State Synchronization Mechanism

Building on the protocol comparison in Section 2.3 (summarized in Table 2), we adopted SSE for server-to-client communication and REST APIs for client-to-server communication. This combination directly reflects the asymmetry of the platform’s communication patterns: the server continuously notifies clients of state changes such as board updates and turn transitions, whereas messages from clients are primarily event-driven requests, such as shot submissions and match-participation requests. For the server-to-client direction, SSE

was selected for its native browser support, built-in automatic reconnection, and affinity with HTTP-based stateless infrastructure. For the client-to-server direction, REST APIs were selected for their statelessness, broad interoperability, and resource-oriented design, which fits naturally with the entities of the platform, such as boards, shots, and match histories. Accordingly, REST endpoints were defined for these entities and used for request transmission from clients to the server. To maintain consistent state notifications across multiple server processes, we use Redis Pub/Sub as the message broker introduced in Section 2.3 [13]. Board-update events are published to Redis channels, and each server process subscribes to these channels so that identical update notifications can be delivered to clients regardless of which server process they are connected to. If an SSE connection on the client side is temporarily interrupted, the server resynchronizes the state after automatic reconnection by referring to the latest state stored in the database. The flow of automatic reconnection and state synchronization after a communication failure is shown in Figure 5. In this process, not only the latest board state but also the board transitions and shot histories within the current end are retransmitted, allowing the match to continue while preserving the context immediately before the disconnection. This design also enables continuous provision of the input sequences required by AI models that use temporal information, such as RNNs, LSTMs, and Transformers.

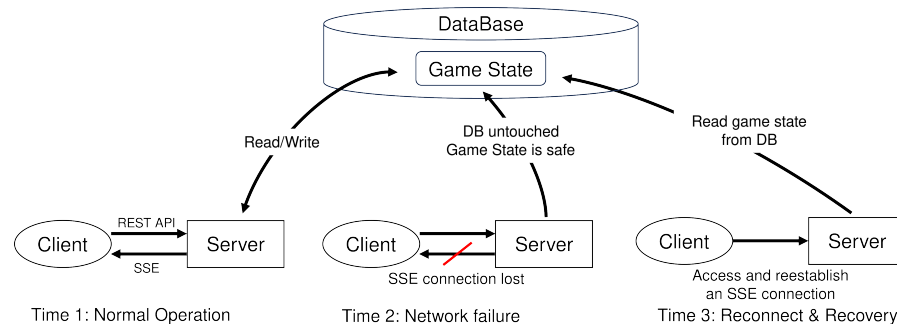


Figure 5. Client-side disconnection recovery mechanism: after a transient communication failure, the client automatically reconnects via the SSE built-in mechanism, and the server resynchronizes the latest board state together with the shot history of the current end.

3.5. Server and Client Implementation

The server was implemented using Python 3.12 and FastAPI [14]. Its internal structure was modularized according to functional responsibilities, including API handling, match control, state transitions, persistence, event delivery, and simulator invocation. The flow of game progression is illustrated in Figure 6. The game progression logic was implemented to handle board updates, turn switching, end-completion judgment, and game-completion judgment. At each turn, the server receives a shot request from the active client, invokes the simulator, and computes the post-shot trajectory and final board state. It then updates the board, turn, and match progression status based on the shot result, stores the updated match information in the database, and notifies connected clients via SSE.

The client was designed as a decision-making module that receives board information from the server, determines the next shot, and sends it back via REST APIs. By adopting an interface independent of the internal AI algorithm, the system can support search-based methods, machine-learning-based methods, and their combinations on a common platform. The event-driven client-side processing and reconnection sequence are shown in Figure 7. If an SSE connection is interrupted, the client automatically attempts to reconnect; once reconnected, the latest match state is resynchronized from the server, and the match continues from the current turn. This mechanism reduces the cost of rerunning matches even in AI-versus-AI games that require long computation times.

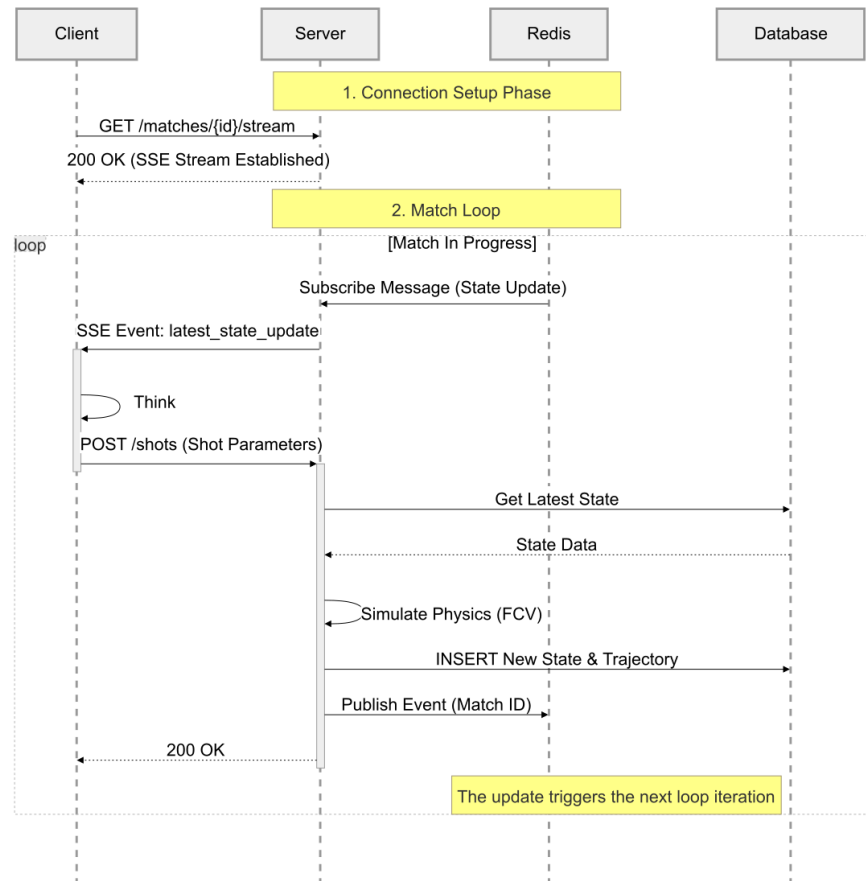


Figure 6. Sequence diagram of game progression in the proposed system, from SSE connection establishment to state persistence and event publication.

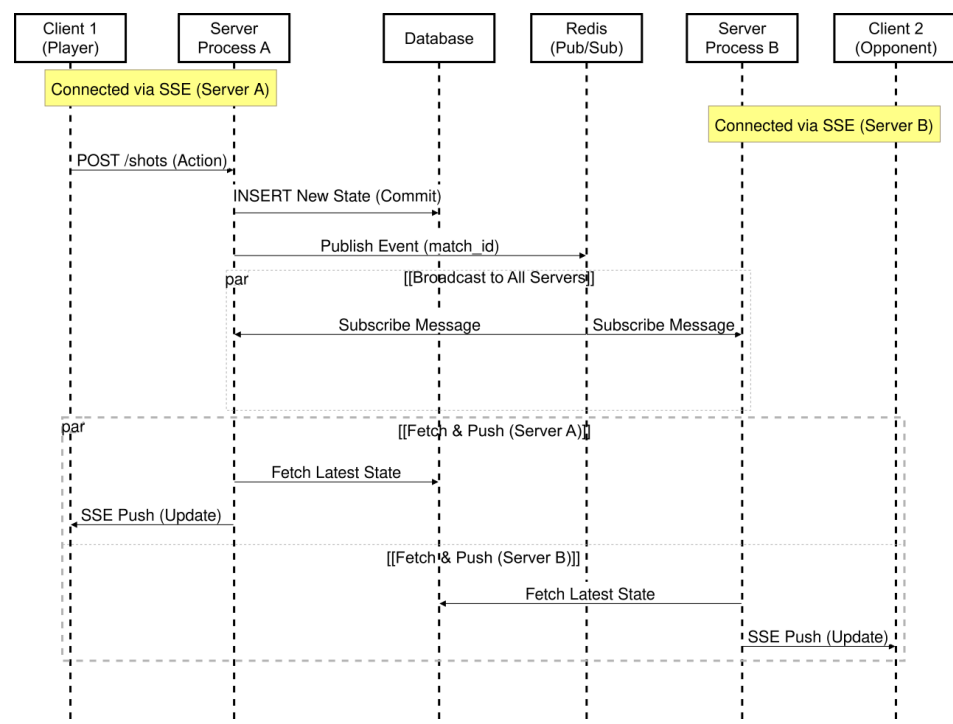


Figure 7. Sequence diagram of client-side processing and reconnection logic, including state retrieval via REST APIs and SSE re-subscription after connection loss.

To improve usability, third-party accessibility, and reproducibility of the proposed platform, we also released the server implementation, the C++ simulator library, a Python client library, and Python client templates for both the four-player format and mixed doubles. The Python client library was implemented as `dc4client` (an abbreviation for “Digital Curling 4th-generation client”) and made available through PyPI. This allows AI developers to focus on implementing shot-selection algorithms without having to build communication and reconnection handling from scratch. The client templates provide project examples for both four-player curling and mixed doubles, each including execution examples with two clients. The configuration files allow users to modify authentication information, match settings, the simulator to be used, applicable rules, and the competition format, making it easy even for first-time users to start a match. By releasing these software components, the proposed system functions not only as a proof of concept in this paper but also as a practically usable research platform for third parties.

3.6. Simulator Decoupling and Support for Multiple Game Formats

The simulator was restructured as an independent library, decoupled from the server. The core physics engine was implemented in C++17 using Box2D as the two-dimensional physics library [15]. The Friction Curl Velocity (FCV) model, adopted in the third-generation Digital Curling system, was employed as the physical model [4,10], and the core simulation logic follows the original implementation. In the process of decoupling the simulator into an independent library, certain iterative routines were refactored and optimized. To enable direct use of the C++ simulator from Python, Python bindings were implemented using `pybind11` [16]. This allows the same simulation engine to be used both for online match execution via the server and for standalone large-scale data generation and model training. To separate the responsibilities of the core physics logic and the external interface, a two-layer architecture was adopted, consisting of the simulation engine itself and a wrapper layer for Python calls. This design preserves computational performance while providing AI developers with an accessible Python interface. In addition, game-rule-specific judgment and restoration routines were implemented as individual functions applied before and after the physics computation, thereby improving adaptability to future rule revisions.

The proposed system is designed to support not only the standard four-player format but also mixed doubles matches. This support was implemented not merely as a configuration option, but across multiple layers: input representation, stone assignment, rule application, and client configuration. The Python interface of the simulator accepts board inputs of 16 stones for the four-player format and 12 stones for mixed doubles. The input format was designed to absorb format-specific differences, with internal conversion to a unified 16-slot representation for processing. Regarding rule application, the system handles the five-rock rule and the no-tick rule for the four-player format, as well as a modified free-guard zone rule for mixed doubles. Furthermore, to account for the pre-placed stone configuration in mixed doubles, an offset was introduced in the stone assignment to distinguish it from the standard delivery-count processing. This enables game states and rule systems that differ across formats to be handled within the same simulation framework. On the client configuration side, the game format can be switched between four-player and mixed doubles, with parameters such as the number of ends, thinking time, applicable rules, and the simulator to be used provided as external configuration. This design allows future additions of game formats or rule updates to be accommodated with minimal changes to the application core.

3.7. Operational Infrastructure via Containerization

The server, database, Redis, and associated services were built and deployed using Docker Desktop 4.66.1 with Docker Engine 29.3.1 [17]. Docker Compose v5.1.0 was used to manage inter-container dependencies and network connectivity [18]. This configuration clearly separated the runtime dependencies of each service component, thereby improving reproducibility between the development and deployment environments. It also enhanced portability and maintainability by facilitating the deployment of the same service configuration across multiple computing environments.

4. Results and Discussion

In this section, we examine the effectiveness of the proposed system from the perspectives of resilience to communication failures and real-world operation.

4.1. Automatic Recovery from Connection Loss and Match Resumption Mechanism

This section evaluates the automatic reconnection mechanism and match continuity of the proposed system under simulated communication failures. As a resilience experiment under communication failures, two clients were connected to a single server, and the system was operated continuously for 24 h under conditions in which communication errors caused by client-side timeouts occurred intermittently. To simulate communication failures during prolonged AI deliberation, the first-player client was configured to suspend its response for an extended period immediately after the start of each match. This caused the opposing client to experience intermittent application-level timeouts while awaiting state updates from the server, thereby enabling continuous observation of automatic reconnection behavior under accidental disconnection. The experimental environment was constructed using Docker on a Windows 11 PC equipped with an AMD Ryzen 7 5800U processor (with integrated Radeon Graphics) and 16 GB of RAM. The application runtime consisted of Python 3.12 on Debian 12 (Bookworm), and PostgreSQL 14.13 was used as the database. The client under evaluation was configured with a reconnection wait time of 1.0 s, consistent with the production environment. No intentional network interruptions were introduced; instead, conditions were set to trigger application-level timeouts, thereby evaluating system behavior under accidental disconnections representative of real operational conditions. The evaluation metrics were the reconnection success rate and the recovery time, defined as the elapsed time from the detection of a `TimeoutError` to the completion of the subsequent board state acquisition.

Analysis of logs collected during the experiment revealed that a total of 290 timeout errors were recorded: 0 for the first-player client and 290 for the second-player client. Automatic reconnection succeeded in all 290 cases, yielding a reconnection success rate of 100%. The mean recovery time from `TimeoutError` detection to reconnection completion was 1008.86 ms, with a standard deviation of 4.57 ms. As shown in Figure 8, the distribution of recovery times was concentrated in the range of 1.00–1.03 s, with no instances of prolonged recovery observed.

These results demonstrate that the proposed system achieves a high reconnection success rate and stable recovery times under the experimental conditions (a single-server, two-client controlled setup with application-layer timeout injection, not live network interruptions), even in the presence of communication failures. Furthermore, the design of persisting the game state at the conclusion of each turn not only enhances match continuity but also serves as an effective foundation for post hoc log analysis and data reuse.

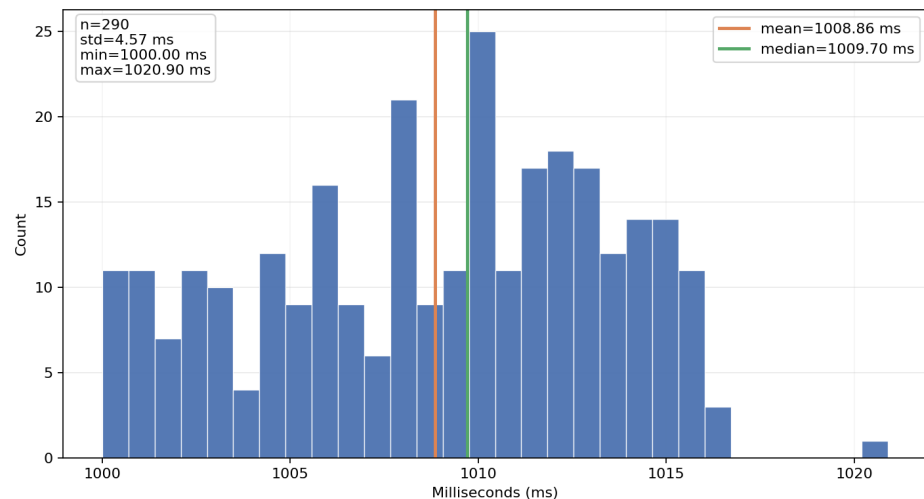


Figure 8. Distribution of recovery times ($N = 290$). Recovery time is concentrated in the 1.00–1.03 s range, dominated by the intentional 1.0 s reconnection wait.

4.2. Recovery from Server Process Failures

While the previous experiment evaluated recovery from transient client-side disconnections, this experiment evaluates the platform’s ability to resume an in-progress match after a complete server-process failure. The key claim under examination is that the database-backed stateless server design decouples match state from server-process lifetime so that a match can be resumed even when the server process is forcibly terminated mid-game.

In the experimental setup, two clients were connected to a single server, and a match was started. The experimental environment was identical to that described in Section 4.1. The experiment was carried out manually; to make it reproducible, each of the $N = 50$ trials followed the same fixed protocol: (1) start a fresh match with two connected clients; (2) allow the match to progress to a pre-selected interruption point, chosen so that terminations were distributed across different ends and shot positions spanning early-, mid-, and late-end phases rather than a single fixed phase; (3) forcibly terminate the running docker compose up process by sending SIGINT (Ctrl+C), stopping all service containers in the stack; (4) restart the stack by re-issuing docker compose up, while the two client processes remained running and continued to attempt automatic reconnection via the SSE built-in mechanism; (5) once the server became responsive, allow the clients to resume from the latest state retrieved from the database and play the match to its conclusion; (6) compare the post-restart state against the last state persisted before termination and record the two outcomes defined below.

For each trial, we recorded two binary outcomes: (i) *Match resumption*: whether the match successfully resumed after the server restarted and proceeded to its conclusion; and (ii) *State consistency*: whether the match state observed after the restart was consistent with the latest state persisted to the database immediately before the termination, in terms of stone coordinates, scores, end and shot counters, and shot history. Recovery time was deliberately not measured in this experiment, as the elapsed time from container shutdown to a fully responsive server is dominated by container startup latency and the manual reissue of the deployment command, neither of which represents the platform’s intrinsic recovery overhead.

The results are summarized in Table 5. Across all 50 trials, both metrics achieved a 100% success rate: every match was successfully resumed after the server restart, and in every trial the state observed after the restart was consistent with the state persisted before termination, with no observed discrepancies in stone coordinates, scores, or turn counters.

Table 5. Results of the server-process failure recovery experiment ($N = 50$).

Metric	Value
Number of trials	50
Match resumption success rate	100% (50/50)
State consistency rate	100% (50/50)

These results confirm that the proposed design—in which all match-progress information is persisted to the database at each shot, and the server is restartable as a stateless process—enables recovery not only from temporary connection losses on the client side but also from full server-process failures. This property is particularly important for long-running AI-versus-AI matches: without database-backed state persistence, an unscheduled server restart would force the entire match to be rerun, wasting substantial computational resources. These results were obtained under controlled laboratory conditions using manually triggered container restarts; production environments with automated orchestration or network-level failures may exhibit different recovery behavior.

4.3. Operational Results in Real-World Environments and Practical Implications

This section examines the stability and practical applicability of the proposed system in a real-world deployment, based on operational experience gained through a public exhibition. A curling experience system was developed using the Digital Curling server, the decoupled simulator, and the client library produced in this study and was exhibited at the Sports Future Lab—Sports Shaping the Future exhibition at Expo 2025 Osaka Kansai. The backend of the exhibition system was built upon the server and libraries developed in this study, with Unity adopted as the client application. The input interface consisted of a stone-shaped controller housing a six-axis IMU sensor and a rotation control motor, along with a brush-type controller capable of detecting sweeping motions, enabling visitors to compete against an AI opponent.

The system was operated over six consecutive days from 3 September to 8 September 2025, during the official opening hours of 10:00–20:00 JST, corresponding to a total of approximately 60 operational hours. No visitor count data were collected at this deployment, and the number of concurrently connected clients was limited due to the single-booth configuration. Throughout this period, no system failures or unresponsive states attributable to the proposed system were observed. These results indicate that the proposed system possesses a degree of stability and operational continuity sufficient for real-world deployment under the tested conditions. Further details on the exhibition system configuration and visitor survey results are reported in [19].

These findings suggest that the proposed system is applicable not only to research use cases but also to interactive applications in real-world environments.

4.4. Load and Scalability Evaluation

To characterize how many concurrent matches the platform can sustain and whether the stateless, database-backed, Redis Pub/Sub design scales across server processes, we drove the system with a load harness built on the official Python client library. Each match is played by two fixed-policy agents that deliver a constant, legal shot on every turn; the agents' strength is irrelevant to this experiment, which measures server capacity rather than AI quality. We ramped the number of concurrent matches $C \in \{1, 2, 5, 10, 20\}$ (i.e., $2C$ player clients), ran each level to completion, and recorded the per-shot round-trip latency from shot submission to receipt of the resulting state, the throughput, and the error rate. We evaluated two configurations: a single server process on one CPU core and two worker processes (via Gunicorn) sharing the same database and Redis channel so that

the multi-process path of the architecture is exercised directly. The load tests were run on a workstation with an Intel Core Ultra 9 285K CPU (3.70 GHz) and 32 GB of RAM.

Table 6 and Figure 9 report the results. Across all matches in both configurations, there were no errors, and every match was completed, so the platform degrades gracefully rather than failing under concurrency. With a single worker, throughput saturates at approximately $26.7 \text{ shots s}^{-1}$ from $C = 2$ onward, corresponding to a serialized per-shot service time of about 37.5 ms (the C++ physics computation plus the state-persistence write). Because one core processes shots at this fixed rate, additional concurrent matches queue rather than speed up, so median latency grows almost linearly with C , from 41.7 ms at $C = 1$ to 728.9 ms at $C = 20$ (a slope of $\approx 36 \text{ ms}$ per added match, consistent with the service time via Little's law). We adopt a 95th-percentile shot-to-state latency of 500 ms as a conservative interactivity target: classical human–computer interaction studies identify 1.0 s as the threshold below which a user's flow of thought stays uninterrupted [20,21], and we set the target at half of that value to leave roughly $2\times$ headroom for network variance and the remaining client-side turn handling so that the complete interaction stays within the 1.0 s limit. This target is conservative for autonomous AI agents, whose per-turn time is dominated by deliberation, and is most relevant to the human-operated, exhibition-style use of the platform. Under this target, a single core sustains roughly $C \approx 10$ concurrent matches.

With two workers, the saturated throughput rises to approximately $52.5 \text{ shots s}^{-1}$, a $1.97\times$ improvement that is close to the ideal $2\times$, and the 95th-percentile latency at a given load roughly halves (for example, $910.5 \rightarrow 526.3 \text{ ms}$ at $C = 20$), extending the 500 ms-objective capacity from $C \approx 10$ to $C \approx 19$. At $C = 1$, the two configurations are indistinguishable, as expected: a single match is turn-sequential and cannot use two workers in parallel; the benefit appears once multiple matches can be distributed across workers. Crucially, the two-worker runs were completed with the same zero error rate and full match completion as the single-worker runs, with no observed state inconsistency, lost matches, or stalls.

Together, these measurements quantify a concrete single-core capacity limit and confirm that the multi-process design behaves as intended. Running additional stateless workers behind the shared database and Redis channel recovers throughput near-linearly while preserving correctness; because no match state is held in any server process, capacity is increased by adding workers rather than by re-architecting. A larger multi-node study—more workers across separate hosts, and the simultaneous mass-reconnection burst discussed in Section 4.5—remains future work, but the two-worker result establishes that the architecture scales as designed.

Table 6. Load and scalability results for one and two server workers. Each row is one level of the concurrency ramp (C concurrent matches = $2C$ player clients). Latency is the shot-to-state round trip at the delivering client. Workers share one database and one Redis channel.

Workers	C	Throughput (Shots/s)	p_{50} (ms)	p_{95} (ms)	p_{99} (ms)	Errors
1	1	23.1	41.7	47.8	49.5	0
	2	26.9	72.6	81.3	106.9	0
	5	26.7	183.7	224.2	256.6	0
	10	26.6	366.4	457.2	520.6	0
	20	26.4	728.9	910.5	1032.8	0

Table 6. Cont.

Workers	C	Throughput (Shots/s)	p ₅₀ (ms)	p ₉₅ (ms)	p ₉₉ (ms)	Errors
2	1	22.7	43.1	47.0	55.9	0
	2	37.6	44.2	74.5	79.4	0
	5	51.7	83.7	143.0	168.6	0
	10	51.5	164.8	306.8	353.2	0
	20	54.3	346.8	526.3	596.8	0

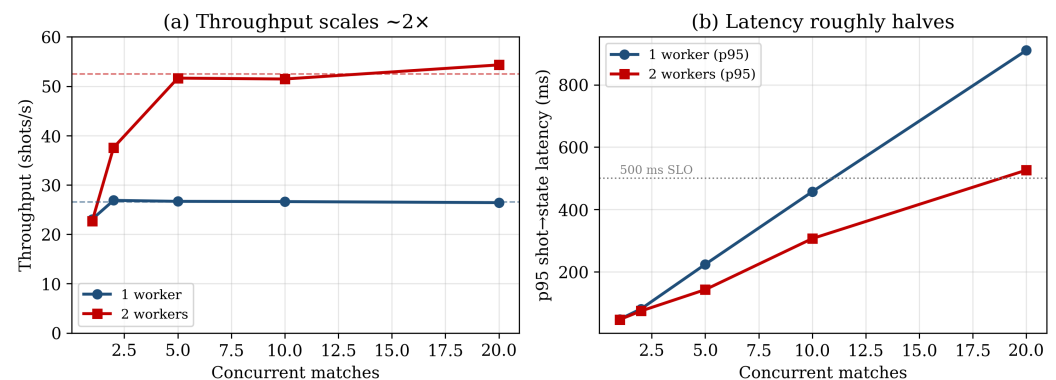


Figure 9. Load test with one versus two server workers sharing one database and Redis channel. (a) Saturated throughput rises from ≈ 26.7 to ≈ 52.5 shots s^{-1} ($1.97\times$). (b) The 95th-percentile shot-to-state latency roughly halves, extending the 500 ms-objective capacity from $C \approx 10$ to $C \approx 19$. No errors occurred at any level in either configuration.

4.5. Limitations and Future Work

Several limitations should be acknowledged. First, the approximately 60 operational hours (six days at 10 h per day) represent a constrained real-world test; longer or higher-intensity deployments may reveal failure modes not observed here. Second, visitor throughput was not instrumented, so the maximum instantaneous concurrent client load during this deployment is unknown. Third, the deployment used a single-server configuration; therefore, the behavior of the Redis Pub/Sub synchronization path under larger-scale multi-server deployments remains future work.

5. Conclusions

In this study, we proposed and implemented a new Digital Curling platform designed with an emphasis on resilience, extensibility, and portability in order to address the limitations of conventional systems, namely the inability to continue matches after communication failures, the dependence of state management on in-memory data, and the tight coupling between the simulator and the server. In the proposed system, the server was designed to be stateless, and matched states were persistently stored in a database. REST APIs were adopted for client-to-server communication, while SSE was employed for server-to-client state notification. Furthermore, by separating and restructuring the simulator as an independent library and making it directly accessible from Python, we established a platform that supports both online match execution and standalone large-scale data generation. The platform also supports not only the standard four-player format but also mixed doubles.

In the resilience evaluation, the system recovered from all 290 timeout-induced communication failures during a 24 h continuous operation experiment, achieving a reconnection success rate of 100%; the mean recovery time was 1008.86 ms, most of which was attributable to the intentionally introduced 1.0 s reconnection wait period. A separate server-

process failure experiment, in which the running service stack was forcibly terminated and restarted at varied points during ongoing matches, also confirmed full recovery in all 50 trials, with complete match-state consistency before and after the restart. In addition, when a curling experience system built on the proposed platform was operated continuously for six days at Expo 2025 Osaka, Kansai, no failures or unresponsive states attributable to the system were observed. These results indicate that the proposed system can continue matches stably across both client-side communication failures and full server-process restarts under the controlled conditions evaluated here, and that it provides not only research utility but also a degree of stability and practicality for interactive exhibition-style applications in real-world environments. Moreover, the design in which the state at the end of each turn is persistently stored was shown to serve not only to improve match continuity but also as a foundation for post hoc log analysis and data reuse.

The scope of these results is bounded in several respects. The resilience experiments ($N = 290$ reconnections; $N = 50$ server restarts) were conducted in controlled single-server, two-client laboratory configurations using application-layer timeout injection and manual container restarts, respectively; they do not directly characterize behavior under live network partitions, hardware failures, or production-scale client loads. The real-world deployment at Expo 2025 Osaka comprised approximately 60 operational hours across six days in a single-booth, limited-concurrency configuration, so the upper bound of sustainable concurrent load remains unknown. In addition, strategic analysis based on the large-scale match logs accumulated on this platform, refinement of strategic evaluation metrics, and the use of the platform as a training infrastructure for high-performance curling strategic AI remain important subjects for future work. Overall, the proposed Digital Curling platform demonstrates resilience to both client-side communication failures and server-process restarts, while also providing extensibility/portability and practical operational suitability. These capabilities are expected to contribute to future Digital Curling research and the advancement of strategic AI for curling.

Supplementary Materials: The following supporting information can be downloaded at: <https://www.mdpi.com/article/10.3390/app16146886/s1>, File S1: full data-definition (DDL) schema; File S2: Docker and Docker Compose configuration files used for deployment.

Author Contributions: Conceptualization, R.K., T.S. and S.A.; methodology, R.K. and T.S.; software, R.K.; validation, R.K.; formal analysis, R.K.; investigation, R.K.; resources, R.K.; data curation, R.K.; writing—original draft preparation, R.K.; writing—review and editing, R.K., T.S., S.A. and T.K.; visualization, R.K.; supervision, T.S. and T.K.; project administration, T.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The software developed and used in this study is openly available on GitHub and archived with persistent, version-specific DOIs on Zenodo. The DigitalCurling4 server is available at <https://github.com/digitalcurling/DigitalCurling4-Server> (accessed on 5 July 2026) (version v1.0.0; <https://doi.org/10.5281/zenodo.20958809>). The C++ FCV simulator library is available at <https://github.com/kr-work/FCV1-Simulator> (accessed on 5 July 2026) (version v1.0.36; <https://doi.org/10.5281/zenodo.20954354>). The Python client library is available at <https://github.com/kr-work/DC4-Python> (accessed on 5 July 2026) (version v1.0.7; <https://doi.org/10.5281/zenodo.20896495>). The Python client templates for the four-player format and mixed doubles are available at <https://github.com/digitalcurling/DigitalCurling4-Client-Python> (accessed on 5 July 2026) (version v1.0.0; <https://doi.org/10.5281/zenodo.20896036>). The Docker and Docker Compose configuration files used for deployment and the full data-definition

schema (DDL; postgres/init/01_init.sql) are included in the server repository and provided as Supplementary Materials; an entity–relationship diagram of the database is shown in Figure 4.

Acknowledgments: This work was supported by the “The Enhancement of HPSC Infrastructure through Technology Innovation Project” of Japan Sports Agency. During the preparation of this manuscript, the authors used Claude (Anthropic, Claude Opus 4.7) for the purposes of language editing, structural review, and manuscript proofreading. The authors have reviewed and edited the output and take full responsibility for the content of this publication.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Kitasei, Y.; Okada, R.; Ito, T. Proposal and Introduction of a Digital Curling Server. *IPSJ J.* **2014**, *2014*, 1–5. Available online: <https://cir.nii.ac.jp/crid/1571135652836403712> (accessed on 5 July 2026).
2. Kitasei, Y. Proposal and Development of a Digital Curling System. Master’s Thesis, The University of Electro-Communications, Tokyo, Japan, 2015. Available online: <https://uec.repo.nii.ac.jp/records/4990> (accessed on 5 July 2026).
3. Mori, K.; Ito, T. Improvement of Digital Curling Robust to Condition Changes. *IPSJ J.* **2019**, *60*, 2085–2092. Available online: <https://cir.nii.ac.jp/crid/1050564288827552128> (accessed on 5 July 2026).
4. Uehara, K. Proposal and Development of an Extensible Third-Generation Digital Curling System. Master’s Thesis, Graduate School of Informatics and Engineering, The University of Electro-Communications, Tokyo, Japan, 24 March 2023. Available online: <https://uec.repo.nii.ac.jp/record/10504/files/2131029.pdf> (accessed on 12 November 2025).
5. Katoh, S.; Iizuka, H.; Yamamoto, M. Development of Curling AI Program Considering the Probabilistic Features of the Strategy. *JSAI Tech. Rep. SIG-KBS* **2015**, *104*, 2. [CrossRef] [PubMed]
6. Katoh, S.; Iizuka, H.; Yamamoto, M. A Method of Game Tree Search in Digital Curling Including Uncertainty. *IPSJ J.* **2016**, *57*, 2354–2364. [CrossRef]
7. Iwasaki, T.; Noguchi, W.; Aihara, S.; Yamamoto, M. Learning of Evaluation Function and Prediction of Winning Probability of Curling Based on Score Difference and Ends Remaining. In *Proceedings of the Symposium on Sports and Human Dynamics*; The Japan Society of Mechanical Engineers: Tokyo, Japan, 2023. [CrossRef]
8. Ohto, K.; Tanaka, T. Applying Monte Carlo Tree Search to Curling AI. In *Proceedings of the Game Programming Workshop 2016*; Information Processing Society of Japan: Tokyo, Japan, 2016; pp. 180–187.
9. Han, Y.; Zhou, Q.; Duan, F. A Game Strategy Model in the Digital Curling System Based on NFSP. *Complex Intell. Syst.* **2022**, *8*, 1857–1863. [CrossRef]
10. Uehara, K.; Ito, T. Developing New Digital Curling System that Is Extendable with Additional Simulation Models. In *Proceedings of the 27th Game Programming Workshop*; Information Processing Society of Japan: Tokyo, Japan, 2022; pp. 61–67.
11. Hassan, M. Choosing the Right Communication Protocol for Your Web Application. *arXiv* **2024**, arXiv:2409.07360. [CrossRef]
12. Kamiński, Ł.; Kozłowski, M.; Sporysz, D.; Wolska, K.; Zaniewski, P.; Roszczyk, R. Comparative Review of Selected Internet Communication Protocols. *Found. Comput. Decis. Sci.* **2023**, *48*, 39–56. [CrossRef]
13. Redis. Redis Pub/Sub. Redis Docs. Available online: <https://redis.io/docs/latest/develop/pubsub/> (accessed on 16 April 2026).
14. Ramirez, S. FastAPI, version 0.115.5, 2024. Available online: <https://github.com/fastapi/fastapi/releases/tag/0.115.5> (accessed on 14 April 2026).
15. Catto, E. Box2D 2.4.1 Documentation, 2021. Available online: https://box2d.org/doc_version_2_4/ (accessed on 14 April 2026).
16. pybind11 Developers. pybind11 Documentation. Available online: <https://pybind11.readthedocs.io/en/stable/> (accessed on 14 April 2026).
17. Docker Inc. What Is Docker? Docker Docs. Available online: <https://docs.docker.com/get-started/docker-overview> (accessed on 16 April 2026).
18. Docker Inc. Docker Compose. Docker Docs. Available online: <https://docs.docker.com/guides/docker-compose/> (accessed on 16 April 2026).
19. Yoshida, S.; Iisaka, C.; Katanuma, M.; Soma, S.; Takahashi, G.; Takubo, Y.; Nakanishi, A.; Muramatsu, R.; Morita, R.; Yamanaka, M.; et al. Miraino Curling: A Curling Experience System Played by Stone Delivery and Sweeping with a Brush. In *Proceedings of the 3rd IPSJ SIG-SI Workshop on Sports Informatics*; Information Processing Society of Japan: Tokyo, Japan, 2025; Paper 15.

20. Miller, R.B. Response Time in Man–Computer Conversational Transactions. In *Proceedings of the AFIPS Fall Joint Computer Conference*; ACM: New York, NY, USA, 1968; pp. 267–277. [[CrossRef](#)]
21. Card, S.K.; Robertson, G.G.; Mackinlay, J.D. The Information Visualizer, an Information Workspace. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '91)*; ACM: New York, NY, USA, 1991; pp. 181–186. [[CrossRef](#)]

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.