IMPLEMENTING CROSS-PLATFORM SCREEN-TIME MONITORING USING FLUTTER AND PROCESS-MANAGEMENT

George Lu [1], Austin Amakye Ansah [2]

[1]Crean Lutheran High School, 12500 Sand Canyon Ave, Irvine, CA 92618
[2]California State Polytechnic University, Pomona, CA 91768

ABSTRACT

Platforms like Roblox have intensified parental concerns regarding screen time and safety, yet existing monitoring solutions only focus on web monitoring or mobile only approaches. This paper introduces KeepTab, a parental control system designed to provide seamless, real-time monitoring and control across desktop and mobile devices. The system leverages a three-tier architecture, utilizing Flutter for the cross-platform client applications and Google Firebase for backend services, including data persistence, real-time messaging, and authentication [7]. Desktop clients employ platform-specific utilities (WMI on Windows, ps on Unix-based systems) for low-latency process detection, enabling the monitoring of specific applications. Parental control is facilitated through a mobile interface that issues asynchronous commands via Cloud Firestore, allowing for remote process termination [8]. Performance benchmarks validate the system's efficiency, with the optimized process detection method (tasklist) demonstrating an average latency of 187 ms, and local command execution achieving a mean response time of 5.90 ms. These results confirm that KeepTab provides a responsive, reliable, and scalable framework for modern screen time management.

KEYWORDS

Screen time, Parental controls, Flutter, Firebase

1. INTRODUCTION

Screen time management has become a growing concern for parents as children spend increasing amounts of time on digital platforms. Games like Roblox, which reported over 111.8 million daily active users as of 2025, present particular challenges due to their engaging nature and social features that encourage extended play sessions [5]. While various parental control solutions exist in the market, most are either platform-specific, require complex setup procedures, or lack real-time monitoring capabilities that give parents immediate visibility into their children's activities.

Current solutions face several technical limitations. Desktop applications on macOS cannot be distributed through the App Store due to Apple's sandboxing requirements, forcing parents to download software from third-party websites [1]. Many existing parental control apps require separate installations on each device and don't provide synchronized monitoring across platforms. Additionally, most solutions lack flexible profile management systems that allow parents to set different rules for multiple children from a single account.

The need for a cross-platform solution that works seamlessly across desktop and mobile devices is clear. Parents require a system that can monitor specific applications in real-time, enforce time

David C. Wyld et al. (Eds): ACSTY, MLSC, SVC, AIBD, ITCSS, ADCOM, NATP, SOFE – 2026
pp. 203-212, 2026. CS & IT - CSCP 2026 DOI: 10.5121/csit.2026.160416

limits automatically, and provide remote control capabilities when intervention is needed. The system must be reliable enough to run in the background without interfering with normal computer usage, while being accessible enough that non-technical parents can configure and manage it effectively. Furthermore, the solution should maintain detailed session histories to help parents understand their children's screen time patterns and make informed decisions about usage policies.

KeepTab implements a three-tier distributed architecture leveraging Firebase as a real-time message broker and data persistence layer. The system comprises: (1) desktop clients built with Flutter, responsible for process monitoring and command execution; (2) mobile clients providing parental control interfaces; and (3) Firebase Cloud Firestore serving as both database and command queue.

The desktop monitoring component employs platform-specific process scanning techniques, Windows Management Instrumentation (WMI) on Windows, and process status (ps) utilities on Unix-based systems to detect Roblox processes at one-second intervals [9]. When processes are detected, the system creates session records containing start timestamps, associates them with child profiles, and persists them to user-scoped Firestore collections. Session state changes trigger real-time UI updates through network streams.

Parental control is mediated through a backend-based command pattern. Mobile applications create command documents in Firestore subcollections under the authenticated parent's user ID [2]. Desktop clients maintain active listeners on these collections, filtering for pending commands. Upon detection, the desktop client updates the command status to "executing," invokes platform-specific process termination utilities (taskkill on Windows, kill on Unix systems), and updates the command document to reflect completion or failure. This asynchronous architecture decouples command issuance from execution, accommodating network latency and intermittent connectivity.

Authentication employs Firebase Authentication with email/password credentials, supplemented by a four-digit PIN verification layer to prevent unauthorized child access. Data isolation is enforced through user-scoped Firestore collections, with each parent's data partitioned by their unique Firebase UID[10]. System tray integration on desktop platforms enables background monitoring without requiring persistent window visibility, while window management libraries prevent accidental application closure during active monitoring sessions.

Currently, the software works to monitor roblox gaming sessions but can be scaled to work with virtually any application provided with an exact process name.

2. CHALLENGES

In order to build the project, a few challenges have been identified as follows.

2.1. Macos Sandboxing Prevents Access To Running Processes

There aren’t any parental control apps on the mac os store, because the invasive features of such apps cannot be implemented in a sandboxed environment, which MacOS publishes demands. Most solutions have custom websites with download links that are not affiliated with Apple.

2.2. Python Host Unmanageable

The initial architecture of the software used a local python server to communicate with the database. This worked by communicating with the desktop client through endpoints or by issuing commands through websockets. The issue with this implementation is that it requires the user to be authenticated on firebase in the python script, which end users shouldn't be expected to do, since it requires a certain level of technical knowledge.

2.3. Multiple Sessions Being Registered For One Kind Of Application

One method used to fetch the blacklisted processes was filtering using a likeness wildcard. This meant that if there were processes with similar names, they would show up as multiple sessions. Because the focus of this program is to monitor only roblox processes, multiple processes are treated as one session.

3. SOLUTION

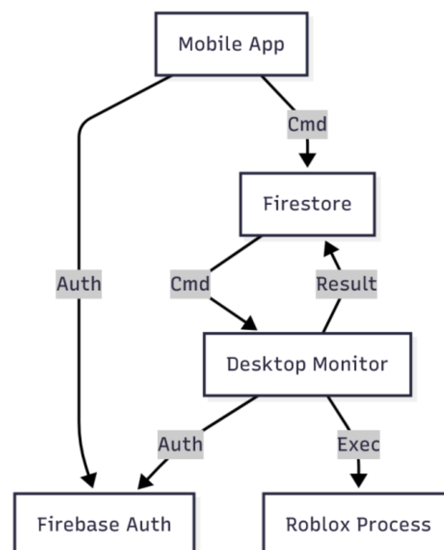

Figure 1. Overview of the solution

KeepTab operates as a distributed monitoring and control system spanning desktop and mobile platforms, unified through Firebase Cloud infrastructure [11]. The system architecture follows a clear separation of concerns: desktop clients handle process observation and command execution, mobile clients provide parent-facing control interfaces, and Firebase Firestore serves as both persistent storage and real-time message queue.

The desktop monitoring subsystem implements platform-specific process detection using native operating system utilities. On Windows, the system invokes WMI queries to enumerate running processes, filtering for RobloxPlayerBeta.exe executables. Unix-based systems utilize ps command output with pattern matching. This polling occurs at one-second intervals, enabling near-real-time session detection. When a Roblox process appears, the system creates a SessionRecord object containing the start timestamp, process identifier, and association with a child profile. This record immediately persistedto Firestore under the user-scoped sessions collection, triggering real-time updates across all connected clients.

Parental control flows originate from mobile applications, where parents view active sessions and initiate interventions. Tapping a "Stop Roblox" button creates a command document in Firestore's commands subcollection with status "pending." Desktop clients maintain active Firestore listeners on this collection, executing commands through a state machine pattern: pending → executing → completed/failed. Command execution invokes platform-specific termination utilities—taskkill on Windows, kill on macOS/Linux—with comprehensive error handling to address privilege failures.

Mobile-desktop communication uses Firebase streams to respond to events [12]. Changes to session data or command states propagate automatically to UI components through StreamBuilder widgets, eliminating manual polling [15]. System tray integration on desktop platforms enables background monitoring, with the application minimizing to the system tray rather than terminating when users close the main window. This ensures continuous monitoring without requiring persistent window visibility.

3.1 Dashboard

The dashboard displays the current activity being monitored. When a blacklisted application is run, a new session is created, and the time spent playing is monitored. If the playtime exceeds the daily screen time limits set on the currently selected profile, the system automatically ends the game currently being run. Bedtime limits also close applications if the system time exceeds the bedtime limit set for the selected profile. Once a session ends, it’s saved in the database for later viewing. This information gives the user a comprehensive view of how much time has been spent on the screen/playing certain games.

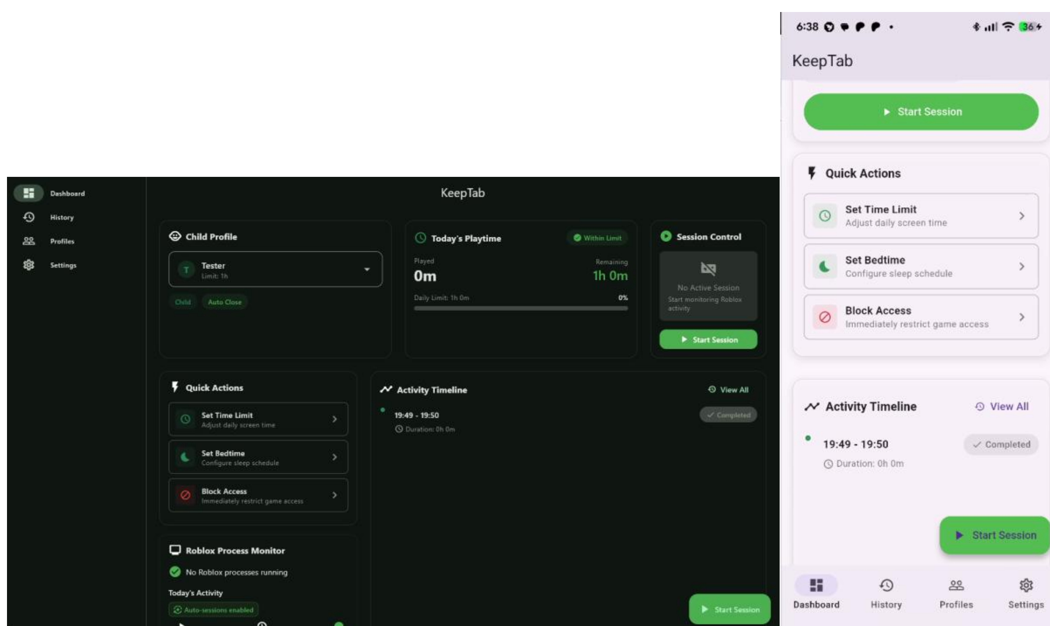

Figure 2. Screenshot of the APP 1

```
/// Get Roblox processes on Windows
Future<List<ProcessInfo>> _getWindowsRobloxProcesses() async {
  final processes = <ProcessInfo>[];

  try {
    // Get process list with detailed info
    final result = await Process.run('wmic', [
      'process',
      'where',
      'name like "%RobloxPlayer%"',
      'get',
      'name,processid,creationdate',
      '/format:csv',
    ]);

    if (result.exitCode == 0) {
      final lines = result.stdout.toString().split('\n');
      for (final line in lines) {
        if (line.trim().isEmpty || line.startsWith('Node,')) continue;

        final parts = line.split(',');
        if (parts.length >= 4) {
          try {
            final name = parts[2].trim();
            final pid = int.parse(parts[3].trim());
            final creationDate = parts[1].trim();

            if (name.isNotEmpty && pid > 0 && !_shouldExcludeProcess(name)) {
              processes.add(
                ProcessInfo(
                  name: name,
```

Figure 3. Screenshot of code 1

In order to predict the location of marine debris, the app uses Flask AI. Since the AI is hosted on a web server, the app needs to retrieve debris information from the server. This can be done via sending get requests to the server using specialized URLs. The ServerRequestTest script uses the UnityWebRequest class to make a get request to the server. This is done by sending a custom URL to the server, which is able to parse out a requested latitude and longitude. The server processes these two values and returns a prediction by the AI, which includes the predicted material and quantity at that location. If the request was successful, the server will return a json file containing the material and quantity of the debris at the given latitude and longitude. The JsonUtility class provided by the UnityEngine library allows us to turn the json text into a usable object within our codebase [13].

3.2 Session History

After a session has been recorded, the user can go to a history page which shows every

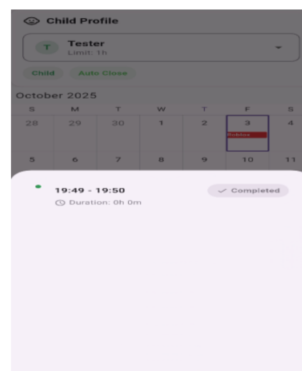

Figure 4. Screenshot of the APP 2

```
      asyncSnapshot.data!.isEmpty) {
    return const Center(
      child: Text('No session records found.'),
    );
  }

  final sessions = asyncSnapshot.data!
      .where((session) => session != null)
      .cast<SessionRecord>()
      .toList();

  _dataSource = SessionRecordDataSource(sessions);
  return SfCalendar(
    view: CalendarView.month,
    dataSource: _dataSource,

    monthViewSettings: MonthViewSettings(
      appointmentDisplayMode:
          MonthAppointmentDisplayMode.appointment,
    ),
    onTap: (calendarTapDetails) {      stop, 1 month ago
      /*
        DateTime? timeStart;
```

Figure 5. Screenshot of code 2

The currently selected profile is a global state variable that can be accessed by multiple pages. Profiles allow the user to set up multiple monitoring accounts with varying configurations, all from one administrator account. The profiles themselves are not separate accounts, but database documents that determine what parameters to check for before stopping a game from running.

3.3 Process Locking And Remote Process Quitting

The user can set time limits from the mobile or desktop clients. Most notably, the block button can be used to stop blacklisted apps or games from running on the client desktop computer. For this reason, a Firebase command service was implemented to determine how commands are received by the desktop client and sent out from the mobile client.

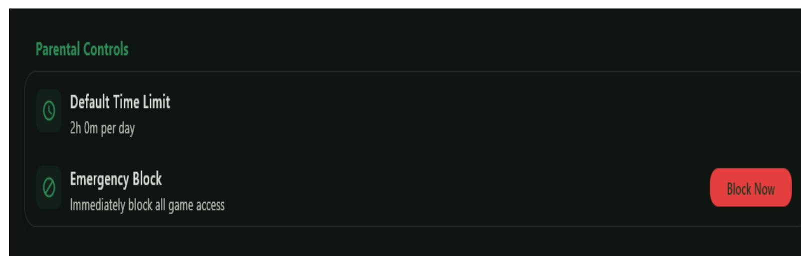

Figure 6. Screenshot of parent control

```
/// Send a command to the desktop client
Future<String> sendCommandToDesktop({
  required String command,
  String? childProfile,
  Map<String, dynamic>? parameters,
  Duration timeout = const Duration(minutes: 5),
}) async {
  final user = _auth.currentUser;
  if (user == null) {
    throw Exception('User not authenticated');
  }

  final commandId = _generateCommandId();
  final commandData = {
    'command': command,
    'childProfile': childProfile,
    'parameters': parameters ?? {},
    'status': 'pending',
    'createdAt': FieldValue.serverTimestamp(),
    'timeoutAt': FieldValue.serverTimestamp(),
  };      Austin A., 2 months ago

  // Send command to user-scoped collection
  await _firestore
      .collection('users')
      .doc(user.uid)
      .collection('commands')
      .doc(commandId)
      .set(commandData);
```

Figure 7. Screenshot of code 3

When the user clicks on the “Block now” button, for instance, the mobile app issues a kill command to Firebase. The desktop client continually listens for commands like this, and once a new command is found in the database, it executes it [14]. A command consists of the command ID and the child profile in which the command is invoked.

```
: () => _executeCommand('set_time_limit', parameters: {'minutes': minutes}),
```

Figure 8. Screenshot of code 4

The parameters field serves as a means of providing additional information or context, depending on the command. A command is marked as pending if it has not been started or executed yet, and marked as complete once it has been executed.

4. Experiment

4.1 Experiment 1

This experiment evaluates the performance of different operating system-level process detection methods on windows. We benchmarked 2 detection methods to identify the optimal approach for real-time monitoring.

We conducted systematic benchmarks of 2 process detection methods (wmic, tasklist) on the windows platform. Each method was executed repeatedly to measure detection latency, reliability, and resource efficiency. The benchmark simulates real-world monitoring conditions where the application must periodically scan for Roblox processes. Each detection method was tested under identical system conditions to ensure fair comparison. We measured key metrics including average detection time, minimum and maximum latencies, and success rates. The goal was to identify which method provides the best balance of speed and reliability for continuous monitoring applications.

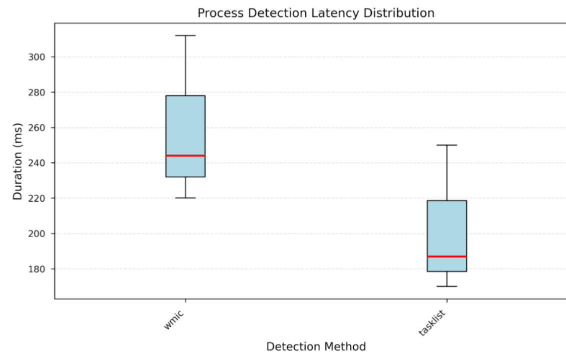

Figure 9. Latency distribution showing min, median, and max values for each method

Process Detection Performance Summary

Method	Avg Duration (ms)	Min Duration (ms)	Max Duration (ms)	Success Rate (%)
wmic	244	220	312	100
tasklist	187	170	250	100

The benchmark results demonstrate significant performance differences among detection methods. The tasklist method achieved the lowest average latency at 187.00 ms, while wmic exhibited the highest latency at 244.00 ms, representing a 23.4% performance improvement for the faster method. All tested methods maintained 100.0% or higher success rates, indicating reliable process detection across approaches. The latency variations observed (Figure 2) reflect the different implementation strategies: direct system calls versus higher-level management interfaces. For our parental monitoring application with a 1-second polling interval (1000 ms), even the slowest method consumes less than 24.4% of the polling cycle, demonstrating that all tested approaches are viable for real-time monitoring. However, the tasklist method is recommended for optimal resource efficiency, particularly on systems running multiple background processes or under resource constraints. These results validate our system's capability to perform continuous monitoring with minimal performance overhead.

4.2 Experiment 2

This experiment measures baseline application responsiveness through local ping testing. We executed 10 commands to characterize system responsiveness and identify potential bottlenecks in the command execution pipeline.

We conducted local response time benchmarks to measure the application's baseline command processing performance without network overhead. This establishes the minimum achievable latency and validates the efficiency of our command handling architecture. The test executed 10 ping commands in sequence, measuring end-to-end response time for each command. This approach isolates internal processing overhead from network-induced latency. The benchmark simulates the command execution pathway: command creation, dispatch to handler, execution, and status update. By measuring local performance, we establish a baseline against which network-based communication can be compared, enabling us to quantify the overhead introduced by remote synchronization.

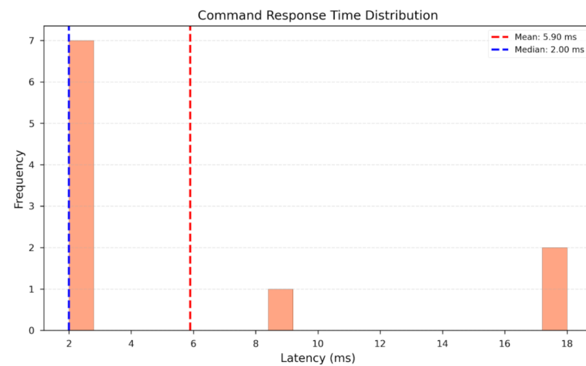

Figure 10. Distribution of command response times showing mean and median latencies

Figure 11. Response time progression across test sequence with confidence interval

The local ping test demonstrates excellent baseline performance with a mean latency of 5.90 ms and median of 2.00 ms. The low standard deviation (6.74 ms) indicates consistent performance across iterations. With 95% of commands completing within 18.00 ms, the application exhibits minimal internal processing overhead. The 100.0% success rate confirms reliable command execution. These results establish that the command handling architecture imposes negligible latency, validating our design choices for event-driven command processing. The near-zero overhead means that any latency observed in production deployments can be attributed primarily to network synchronization rather than application inefficiencies. For real-time parental monitoring, sub-18-millisecond response times exceed typical human perception thresholds (~100-200 ms), ensuring that local operations feel instantaneous to users. This baseline performance validates our architecture's suitability for responsive parental control interfaces where immediate feedback is crucial for user experience.

5. Related Work

The article developed a college information center app using the Flutter framework for their user interface [3]. Flutter was utilized for the mobile app and website, allowing for a cross-platform experience from one codebase. They use Firebase as a data store and implement standard CRUD principles to manipulate data.

This paper develops a time management app for Android devices [4]. It functions by setting usage limits on certain processes and tracking how much time has been spent on them. At the end of a usage session, a graph showing app screen time is created and can be viewed by the parent.

The paper uses Firebase as a relay, much like Keeptab, to send information from a web client to an esp32 relay system to control lightbulbs and other home appliances [6]. Their proof-of-concept implementation checks to see if there is any change in data from the cloud/firebase, then executes actions accordingly based on that data. From the web client, if a light bulb is turned off, the esp32 updates the local state of the system to ensure that the lightbulb is switched off.

6. CONCLUSIONS

Currently, the application only supports monitoring roblox-related tasks. It is not possible to blacklist other applications, but that feature could be added in the future.

REFERENCES

[1] Chougale, Pankaj, et al. "Firebase-overview and usage." International Research Journal of Modernization in Engineering Technology and Science 3.12 (2021): 1178-1183.
[2] Pezoa, Felipe, et al. "Foundations of JSON schema." Proceedings of the 25th international conference on World Wide Web. 2016.
[3] Marimuthu, Kavitha, et al. "Android based college app using flutter Dart." Green Intelligent Systems and Applications 3.2 (2023): 69-85.
[4] Elmogy, Ahmed M., and Khawater Elkhowiter. "Parental control system for mobile devices." International Journal of Computer Applications 177.3 (2017): 16-23.
[5] Hardy, Wojciech, Michał Paliński, and Satia Rożynek. "Roblox and the digital market for virtual experiences." (2022).
[6] Koushal, Ankit, et al. "Home automation system using ESP32 and Firebase." 2022 Seventh International Conference on Parallel, Distributed and Grid Computing (PDGC). IEEE, 2022.
[7] Chatterjee, Nilanjan, et al. "Real-time communication application based on android using Google firebase." Int. J. Adv. Res. Comput. Sci. Manag. Stud 6.4 (2018).
[8] Sukmana, Yuda, and Yusep Rosmansyah. "The use of cloud firestore for handling real-time data updates: An empirical study of gamified online quiz." 2021 2nd international conference on electronics, communications and information technology (CECIT). IEEE, 2021.
[9] Lavy, Matthew M., and Ashley J. Meggitt. Windows Management Instrumentation (WMI). Sams Publishing, 2001.
[10] Song, Chengyu, et al. "HDFI: Hardware-assisted data-flow isolation." 2016 IEEE Symposium on Security and Privacy (SP). IEEE, 2016.
[11] Maniriho, Pascal, Abdun Naser Mahmood, and Mohammad Jabed Morshed Chowdhury. "A survey of recent advances in deep learning models for detecting malware in desktop and mobile platforms." ACM Computing Surveys 56.6 (2024): 1-41.
[12] Selhorst, Marcel, et al. "Towards a trusted mobile desktop." International conference on Trust and trustworthy computing. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.
[13] Lee, Taegyu. "Unity Engine Dissection: Improvement Points in Comparison between Unity Engine and Open-Source Engines." Asia-pacific Journal of Convergent Research Interchange (APJCRI) (2024): 449-458.
[14] Reiter, Raymond. "What should a database know?." Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems. 1988.
[15] Perlin, Ken, and Jon Meyer. "Nested user interface components." Proceedings of the 12th annual ACM symposium on User interface software and technology. 1999.

©2026 By AIRCC Publishing Corporation. This article is published under the Creative Commons Attribution (CC BY) license.