JOURNAL OF
CLIMATE INNOVATION AND TECHNOLOGY (JCIT)
Advancing Research in Climate, Sustainability and Emerging Technologies
CLIMATE | SUSTAINABILITY | INNOVATION | TECHNOLOGY

An Intelligent AI Check-In Framework for Automating International Student Onboarding and Administrative Services

Sahar Bukhari[1], Faty Top[2], Keshauna Bromley-Laird[3], Dr. Aruna M. Jarju[4], Balikis Alarape[5]
King Graduate School, Department of Information Technology, Monroe University, New Rochelle, NY, USA
[1] sbukhari@monroeu.edu, [2] ftop@monroeu.edu, [3] kbromleylaird@monroeu.edu, [4] ajarju@monroeu.edu, [5] balarape@monroeu.edu

Abstract
International and transfer student check-in processes involve complex, multi-step workflows that can be streamlined with artificial intelligence (AI). This paper presents the design and implementation of an AI-driven check-in agent that automates student onboarding tasks in a university setting. The agent integrates a graphical user interface (GUI) with AI capabilities to guide students through login/authentication, document uploads, advisor assignment based on region, course pre-selection, and real-time status tracking. We provide a comprehensive literature review of similar systems in the U.S. and globally, highlighting how institutions have leveraged chatbots and AI for admissions and orientation. The system's architecture is described with a technical focus on its Python-based implementation (Tkinter for GUI, PIL for imaging), web integration through Jupyter Notebook compatibility, and open-source API utilization (including BeeWare for cross-platform deployment). We detail each system component and discuss security measures (compliance with FERPA, GDPR) to protect educational data privacy. Challenges such as user adoption, multilingual support, and integration with cloud services are examined, along with future enhancements like machine learning-driven personalization. The paper concludes with expected benefits in institutional efficiency and student satisfaction, positioning the AI check-in agent as a valuable contribution to smart campus initiatives.

Keywords: *Artificial Intelligence, Neural networks, natural language, natural language, AI-driven solutions*

Introduction

An Artificial Intelligence (AI) agent is a computer system designed to autonomously perceive its environment, make decisions, and take actions to achieve specific goals. In the context of this research, an AI check-in agent serves as a virtual assistant that interacts with international students, guiding them through the administrative and onboarding processes at educational institutions. Neural networks, a subset of machine learning algorithms inspired by the human brain's structure, enable the AI agent to recognize patterns, understand natural language, and adapt to diverse student needs.

By leveraging neural networks, the AI check-in agent can accurately process large amounts of unstructured data, learn from interactions, and provide personalized responses.

Together, natural language and neural networks form the backbone of an intelligent, efficient, and user-friendly system that enhances the experience of international students during their transition to new academic environments.

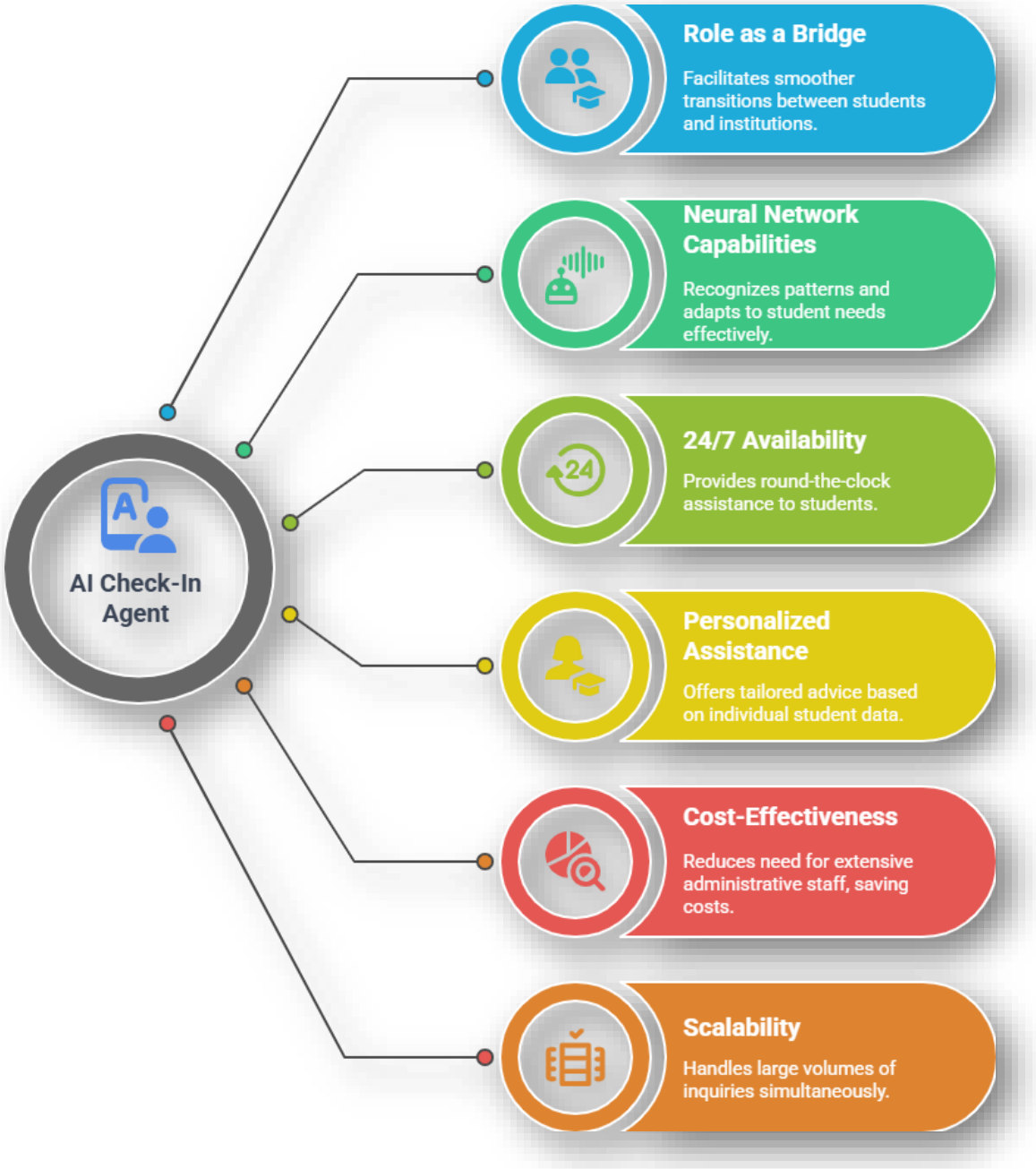

FIGURE 1: Key features and benefits of the AI Check-In Agent, highlighting its role in enhancing admission process support through personalized assistance, neural network capabilities, continuous availability, cost-effectiveness, and scalable service delivery.

Institutions of higher education face increasing pressure to improve the efficiency and user-friendliness of student onboarding, especially for international and transfer students who navigate additional documentation and orientation requirements. Traditional check-in procedures often involve

in-person sessions with staff, manual form filling, and waiting periods. These conventional methods can lead to bottlenecks during peak intake times, causing longer wait times and reduced student satisfaction. To address these challenges, universities have begun exploring AI-driven solutions, such as chatbots and intelligent agents, to automate and enhance the check-in and orientation process. An AI-driven check-in agent can operate 24/7 without the constraints of office hours, provide instant responses to common inquiries, and guide students through required tasks step-by-step.

Recent advancements in natural language processing and machine learning have enabled chatbots to handle a wide range of student services queries with high accuracy. In admissions and onboarding contexts, AI chatbots have been shown to improve student engagement and outcomes. For example, Georgia State University's "Pounce" chatbot reached out to incoming students via text, reminding them of outstanding tasks (like submitting transcripts) and answering their questions, resulting in a measurable increase in enrollment yield. Pounce's machine-learning powered conversations handled the majority of student questions automatically, freeing up staff to address only the most complex issues. This success underscores how AI agents can "work smarter" alongside university staff. Globally, institutions have implemented similar tools: Nguyen et al. (2021) developed an admissions chatbot for Vietnam's National Economics University using the Rasa framework, achieving 97.1% answer accuracy on test queries. Aloqayli and Abdelhafez (2023) built an intelligent admissions chatbot using a third-party platform (Botsify) that correctly answered student questions with 91% accuracy in a controlled evaluation. These systems demonstrate the potential for AI-driven agents to streamline university processes such as orientation, document verification, and advising. However, many existing solutions focus on conversational Q&A through messaging platforms; fewer have tackled an end-to-end guided check-in workflow through an interactive GUI, which is the focus of our work.

This paper contributes a generalized design and implementation of an AI-driven check-in agent for international and transfer students. Distinct from pure chat interfaces, our agent combines a graphical application interface with AI-backed logic to handle structured tasks (e.g. uploading visa documents) as well as potential conversational integration for FAQs. The agent is implemented in Python, leveraging a lightweight GUI (Tkinter) suitable for both kiosk deployment (e.g. on a tablet in an admissions office) and web-based access via Jupyter environments. By using open-source libraries and APIs, the system remains extensible and reproducible for other institutions. The following sections provide a literature review of related systems, detail the system architecture and technical components, discuss security and privacy considerations, evaluate current limitations, and outline future improvements. We aim to provide both a blueprint and a proof-of-concept implementation that campus IT developers and researchers can build upon to enhance student check-in experiences.

Literature Review

AI Chatbots in University Onboarding: Universities worldwide are increasingly adopting AI chatbots to assist with admissions, orientation, and student support. In the United States, one notable early adopter was Georgia State University with its chatbot "Pounce," which was designed to combat "summer melt" – the phenomenon of accepted students failing to enroll. Pounce, developed in partnership with an ed-tech startup, engaged students via SMS to remind them of tasks and deadlines and to answer questions about financial aid, housing, and other pre-matriculation steps. A controlled trial showed that students who received Pounce's AI-guided

assistance was 3.3% more likely to enroll on time compared to a control group. By automatically addressing routine inquiries, the chatbot significantly reduced the workload on admissions staff and allowed them to focus on complex cases. Following this success, many U.S. institutions rolled out similar chatbot systems for admissions and new student orientation (often through vendors or platforms like Mainstay, formerly AdmitHub, and Ocelot). These systems demonstrate improved communication with incoming students but typically function through text-based chat interfaces rather than a dedicated multi-step check-in application.

Globally, AI-driven student support has likewise gained traction. In Vietnam, Nguyen et al. introduced "NEU-Chatbot" to answer admission queries at National Economics University. Using Rasa (an open-source conversational AI framework), their chatbot parsed student questions for intents and entities and responded with relevant information about curricula, fees, and admissions procedures. The NEU-Chatbot achieved an accuracy of over 97% on test questions, though it required manual updates each academic year for new content and had to address occasional misinterpretation of intents. In Morocco, researchers prototyped a "Chatbot E-Orientation" to guide students in educational and vocational choices, highlighting the international interest in AI for student guidance. Meanwhile, in Saudi Arabia, Aloqayli and Abdelhafez (2023) deployed an admissions chatbot at Princess Nourah University, measuring both accuracy and user satisfaction: their system correctly answered student inquiries with 91% accuracy and scored 76.6 out of 100 in a usability questionnaire, indicating generally positive student experiences.

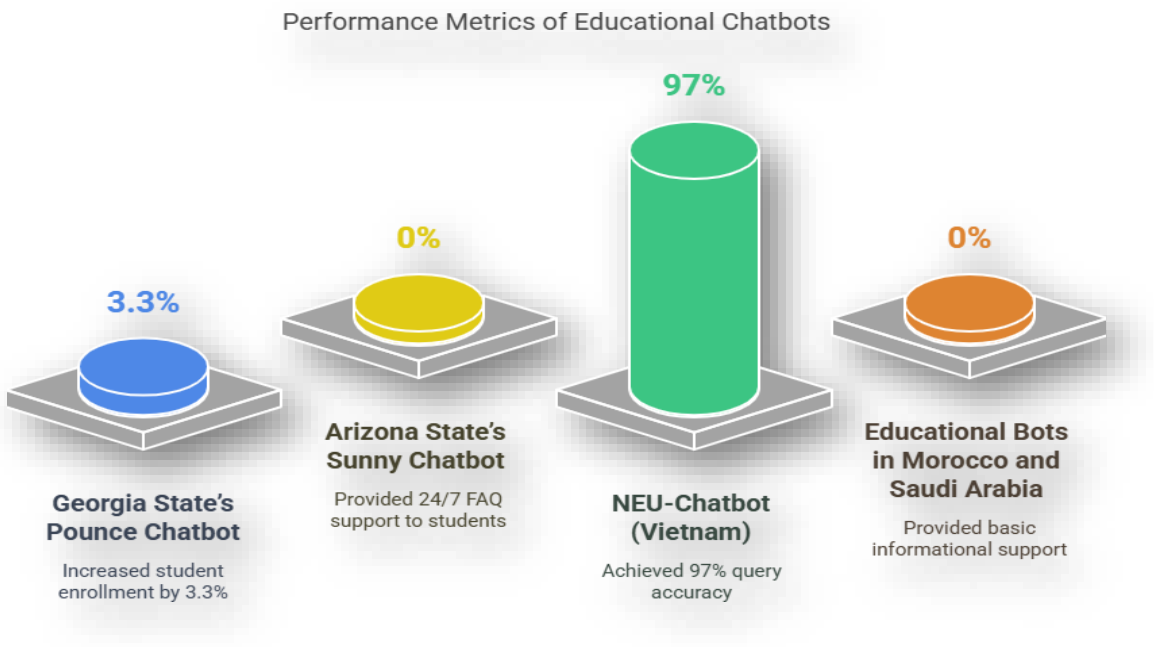

Figure 2: Comparative performance metrics of educational chatbots implemented in higher education institutions, illustrating their impact on student enrollment, query accuracy, service availability, and institutional support across different countries.

These global implementations primarily focus on answering FAQs and providing information, often through web chat or messaging apps, complementing or replacing traditional face-to-face advising.

The literature also reveals studies comparing platforms and methods for educational chatbots. Smutny and Schreiberova (2020) reviewed 89 educational chatbots on Facebook Messenger and found that 89% communicated in English, and 46% relied purely on predefined answers without sophisticated dialogue management. Their review highlighted that many early chatbots were limited in conversational ability, often lacking deep interactive techniques or multi-turn context handling. Other researchers have catalogued challenges in implementing chatbots in education. Okonkwo and Ade-Ibijola (2019) identified common issues including technical complexity in programming the bot, users' trust and attitude towards AI, and difficulties in evaluating chatbot effectiveness.

Ethical and privacy concerns are especially pertinent in education, as noted by Ruane and Birhane (2019), who argue that chatbot functions should be transparent and that users must know how their data and interactions are being used. These findings underscore that beyond achieving functional accuracy, an AI agent for student services must be user-friendly, trustworthy, and compliant with data privacy standards. In contrast to general-purpose chatbots, an AI check-in agent handles sensitive personal data (passports, visas, academic records) and coordinates actions (like uploading documents and assigning advisors), which raises the bar for security, privacy, and reliability.

Existing Check-In Systems:
Outside the domain of chatbots, many universities employ specialized software for international student check-in and management. A prominent example is Terra Dotta, a commercial platform used by international student offices for managing visa compliance, document uploads, and orientation workflows. Terra Dotta provides a web portal where students upload their immigration documents (e.g. visa, I-94, I-20), which staff then verify. It also allows tracking of student statuses through various stages (pre-arrival, check-in, post-arrival) and integration with SEVIS for regulatory compliance. However, these systems typically do not incorporate AI; the interaction is form-based and requires students to navigate a series of web pages. While effective for record-keeping and compliance, such software may lack the conversational guidance and personalized feedback that an AI agent can provide. Our approach seeks to bridge the gap by building a system with the robustness of structured workflows (similar to Terra Dotta's document and status tracking) combined with the interactivity and intelligence of chatbots (similar to Pounce or NEU-Chatbot). By reviewing both chatbot implementations and check-in management systems, we inform the design of an integrated agent that can handle structured tasks and unstructured queries in the check-in context.

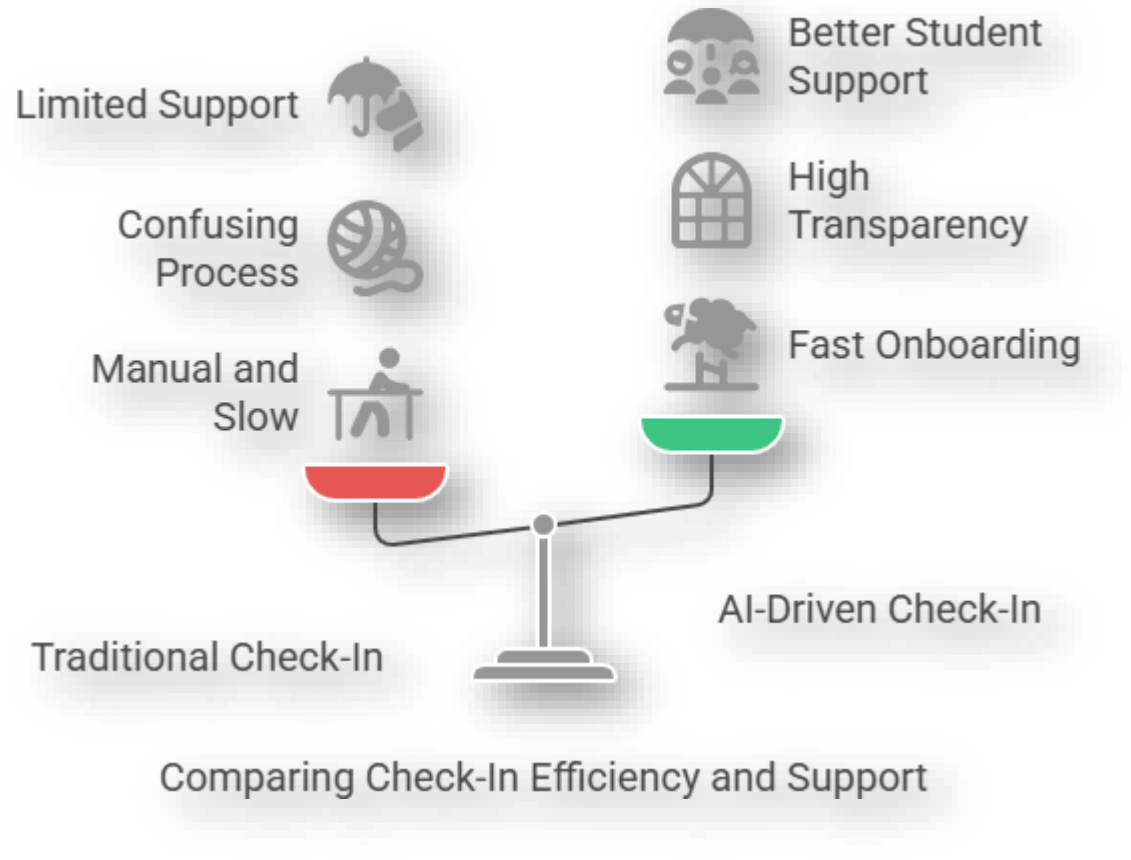

Figure 3: Comparison of traditional and AI-driven student check-in process, illustrating how artificial intelligence enhances student support, increases transparency, accelerates onboarding, and improves the overall efficiency of the student check-in process.

System Design and Architecture

Overview: The AI-driven check-in agent is designed as a multi-component system combining a front-end interface with back-end logic and potential AI services. The high-level architecture follows a client-agent model: the client side (running on a kiosk tablet or a student's web browser) presents a GUI for user interaction, while the agent's logic can interface with databases and APIs to process inputs and generate responses.

Each step is encapsulated in a module within the application. The design ensures that progression can be non-linear when needed (for instance, a student might log back in later to upload additional documents or change course selections, and the agent will retrieve their previous status). This modular approach aligns with typical check-in processes defined by university international offices
, which often segment tasks into authentication, orientation, document submission, and advising.

On the back end, the system can integrate with existing university infrastructure. For example, the login module can be linked to the university's single sign-on service or student information system (to validate student IDs). The document upload module could interface with cloud storage or a content management system (such as an Azure blob storage or Google Drive API) to save and scan the documents. Advisor assignment may pull data from a registrar or advising office database

mapping regions to advisor names. Course pre-selection could connect to the university's course registration system or at least retrieve a list of available first-semester courses for the student's program. In our implemented prototype, for simplicity, these integrations were mocked or hard-coded (e.g., using a preset list of courses and a fixed mapping of regions to advisors), but the design anticipates replacement of these with API calls or database queries in a production deployment. The agent's AI capabilities can be expanded by incorporating a conversational component using an NLP service (such as Google Dialogflow or an open-source framework). For instance, while the current UI flows are menu-driven, an AI chatbot could be embedded to allow students to ask natural language questions at any point (e.g., "What documents do I need to upload?") and receive answers drawn from an FAQ knowledge base. This hybrid design (graphical workflow + conversational assistance) would combine structured guidance with AI flexibility, enhancing user experience.

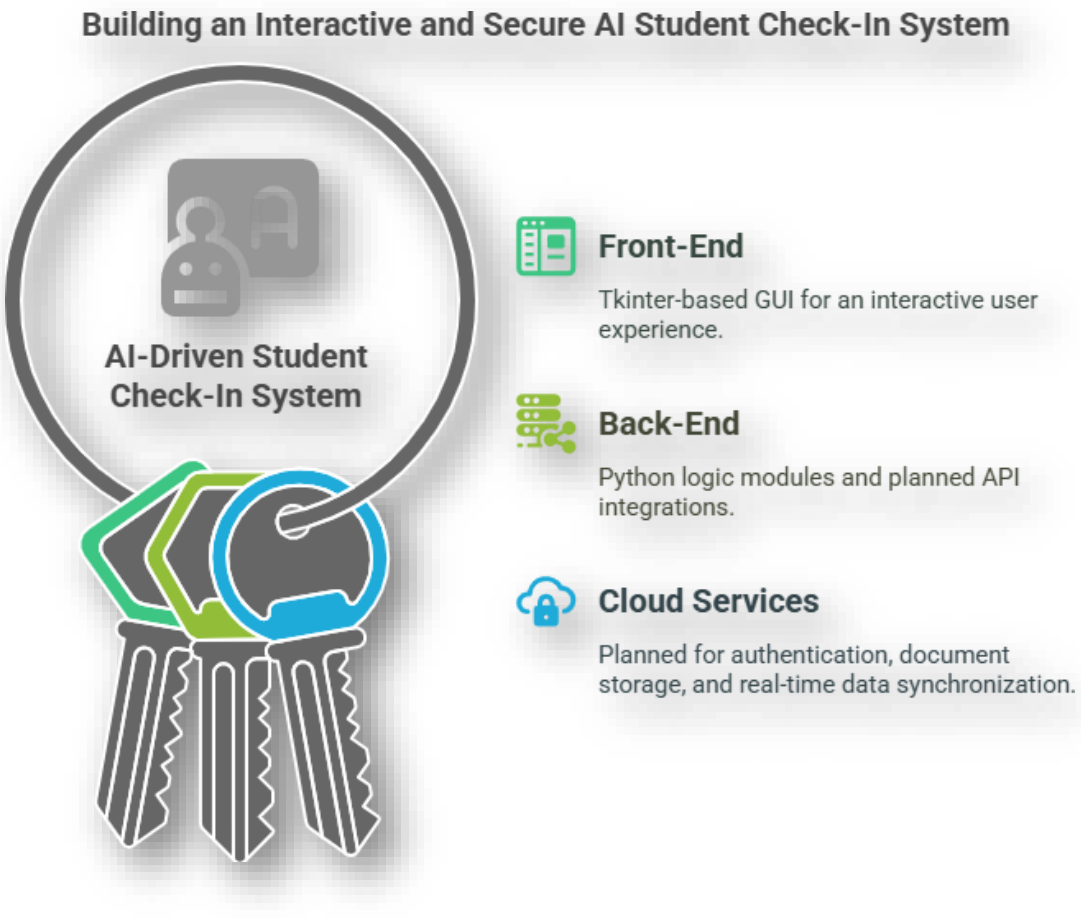

Figure 4: Architecture of the AI-Driven Student Check-In Process, illustrating the integration of the front-end interface, back-end processing, and cloud services to provide secure, scalable, and efficient student check-in and onboarding.

The choice of technologies in the design emphasizes accessibility and openness. Python was selected as the implementation language due to its rich ecosystem of libraries for GUI development, image processing, web integration, and machine learning. By using Python, the project can easily leverage open-source packages and frameworks, and developers worldwide (including academic researchers) can reproduce or extend the agent without licensing barriers. The GUI is built with Tkinter, Python's standard GUI toolkit, which offers a lightweight interface suitable for desktop and can be executed in cross-platform environments. For image handling (such as loading a university logo and processing uploaded images), we use PIL/Pillow. The entire application can run as a local program or within a Jupyter Notebook context for demonstration purposes, thanks to Python's interactivity. This means a university could prototype the agent in a Jupyter environment, making it accessible via a web browser for testing without a complex web server setup. Furthermore, to deploy the agent beyond just a PC, we integrate BeeWare's Briefcase tool, which packages Python applications into standalone executables for multiple platforms. Using Briefcase (an open-source packaging tool), the check-in agent can be converted into a native mobile app or a desktop application for Windows, macOS, or Linux, distributing the AI agent conveniently to end-users such as students' smartphones or on dedicated tablets. By adhering to open standards and tools, our system design maximizes compatibility and ease of integration with other systems.

User Interface Flow: The check-in agent's interface is organized into sequential screens that correspond to each major task in the check-in process. The design follows a guided workflow, using clear prompts and visual cues so that even students unfamiliar with the system can navigate it easily. The interface uses a consistent color scheme and branding (which can be customized for any institution) to create a welcoming experience; for example, the prototype uses a blue theme (background #1a237e) with contrasting text for readability, and includes a university logo at the top of each screen (or a placeholder title if the logo file is unavailable). Navigation buttons ("Next", "Back", etc.) are placed intuitively to move between steps, and a persistent footer shows the university's name and year to reinforce credibility.

When the application launches, it starts at the **Login Screen**. This screen asks the student to enter their Student ID in a text entry field. We implemented a simple login for demonstration, where any input moves to the next step (no back-end authentication in the prototype), but in practice this would be tied into an authentication system. The login screen design uses a prominent title "Student Login" and the university logo at top. The student ID entry field is centered, and a **Login** button triggers the authentication check. By using Tkinter's `Entry` widget and `Button` with callback functions, the application captures user input and responds to the button click by invoking the next screen's function (if credentials are valid). Notably, we added basic GUI enhancements such as solid border styles and padding to make the UI elements accessible and visually consistent. The use of a GUI framework means we can also provide immediate feedback – for example, if login fails (in a full implementation), a pop-up message could notify the student to retry or contact support.

After successful login, the agent proceeds to the **Region Selection Screen**. International and transfer students often need to meet or be assigned to specific advisors or student coordinators based on their region or country of origin. Our system streamlines this by asking the student to indicate their home

region from a drop-down menu. In our prototype, we defined example regions (e.g., "Nepal", "South East Asia", "Bangladesh", "China/Taiwan/South Korea"), but this list can be configured to match an institution's advising assignments (for instance, regions might correspond to continents or specific programs). The interface uses Tkinter's OptionMenu for the drop-down selection bound to a variable that tracks the current choice. Once the student selects a region, they click a **Submit** button to confirm. The agent then looks up the advisor associated with that region. This logic is implemented with a simple conditional mapping: e.g., if "Nepal" then advisor = "Shruti"; if "South East Asia" then advisor = "Anna K. Dennis"; etc. This mapping could alternatively be a dictionary or database query in a real system. If a region was selected, the agent triggers an info dialog (using Tkinter's message box) to inform the student: "Your advisor is [Name]". Immediately after, the agent navigates to the check-in progress screen. If no region was selected (e.g., the student hit Submit without choosing), the agent instead shows a warning message prompting the user to make a selection. This validation ensures the student cannot accidentally skip this critical step. The region selection screen also provides a Back button (to return to login in case the student needs to re-enter credentials or switch accounts) and includes the standard footer. By assigning the advisor at this stage, the system personalizes the experience— subsequent screens can display the advisor's name, and the information can be logged for staff to know which students have been assigned to whom.

Following advisor assignment, the **Check-In Progress Screen** serves as the home dashboard for the student's check-in status. This screen aggregates the status of various tasks and allows navigation to complete them. It is conceptually similar to a student portal page showing one's to-do list. In our implementation, this screen shows labels for each major item: Document Status and Advisor, and provides action buttons for the remaining tasks (upload documents, pre-select courses). Initially, the Document Status is shown as "Pending" to indicate the student still needs to upload required files, and the Advisor label displays the name obtained from the previous step. The progress screen will be revisited later, potentially updated once documents are uploaded or other steps finished. From this dashboard, if the student clicks **Upload Documents**, the agent opens the Document Upload interface; if they click **Pre-Select Courses**, it opens the course selection interface. This hub-and-spoke design allows the student to tackle tasks in any order (after advisor assignment) and return to the status page. The check-in progress screen remains accessible until all tasks are done, at which point the process would be considered complete (the prototype simply allows tasks to be revisited indefinitely since we did not implement a completion rule).

The **Document Upload Screen** addresses the requirement for international students to submit paperwork such as transcripts, visa/I-94 copies, immunization records, etc. This screen presents a button to "Upload Document" which, when clicked, opens a file dialog (using filedialog.askopenfilename) to let the student choose a file from their device. We integrated Python's tkinter.filedialog for this purpose, which is a simple way to access the local filesystem and select files. Once a file path is returned by the dialog, our code simulates an upload action by immediately showing a confirmation dialog with the file name (messagebox.showinfo("Uploaded", f"Document uploaded: {file_path}")). In a real deployment, this step would involve actually transferring the file to secure storage on a server or cloud service and perhaps virus-scanning and verifying the file type. The GUI could also display the list of files uploaded so far and their verification status. For example, after uploading, the status might change from "Pending" to "Submitted" or "Verified" once approved by an officer. Our prototype keeps it simple, but it lays the groundwork for integrating with, say, an API endpoint that accepts file uploads via

HTTP. The document upload screen also contains a Back button which returns the student to the Check-In Progress screen, allowing them to proceed to other tasks or log out if they wish to pause. By isolating the upload function on its own screen, we ensure the process is user-friendly – the student deals with one file at a time in a clear manner, reducing confusion that might arise if multiple forms were handled in one view. Moreover, this design could be extended: for example, multiple upload buttons or instructions for different document types could be added (passport, visa, health forms each with separate prompts), improving clarity on what needs to be uploaded.

The **Course Pre-Selection Screen** provides an interface for students to pick initial courses before meeting an academic advisor. Many universities encourage new international students to indicate course preferences or complete a preliminary registration as part of check-in, so that they can later finalize schedules with an advisor. Our interface for this feature presents a drop-down menu of courses and a submit action. In the prototype, we used a placeholder list of course names ("Course 1", "Course 2", "Course 3"). These would be dynamically generated or fetched from a database in a real system, likely based on the student's program or intended major.

Using Tkinter's OptionMenu (similar to the region selection), the student can choose one course at a time. We kept the UI minimal: a label "Select Courses" and one drop-down for a course choice. A more advanced version might allow selecting multiple courses or ranking preferences. When the student hits **Submit Selection**, the system currently just displays a success message ("Course selected") using a messagebox, but in practice this would record the selection in a backend (e.g., update the student's registration request record). After submitting, the student can use the Back button (provided via a utility function create_back_button) to return to the main progress screen.

In a full implementation, once courses are selected, the Check-In Progress screen might show an updated status (e.g., "Course Pre-Selection: Done") and possibly prevent resubmission unless changes are allowed. Our design thus supports an iterative improvement: the logic for course selection is segregated, making it easy to replace the dummy functionality with actual registration integration (such as calling a university's course enrollment API or storing choices in a file or database).

JOURNAL OF
CLIMATE INNOVATION AND TECHNOLOGY (JCIT)
Advancing Research in Climate, Sustainability and Emerging Technologies
CLIMATE | SUSTAINABILITY | INNOVATION | TECHNOLOGY

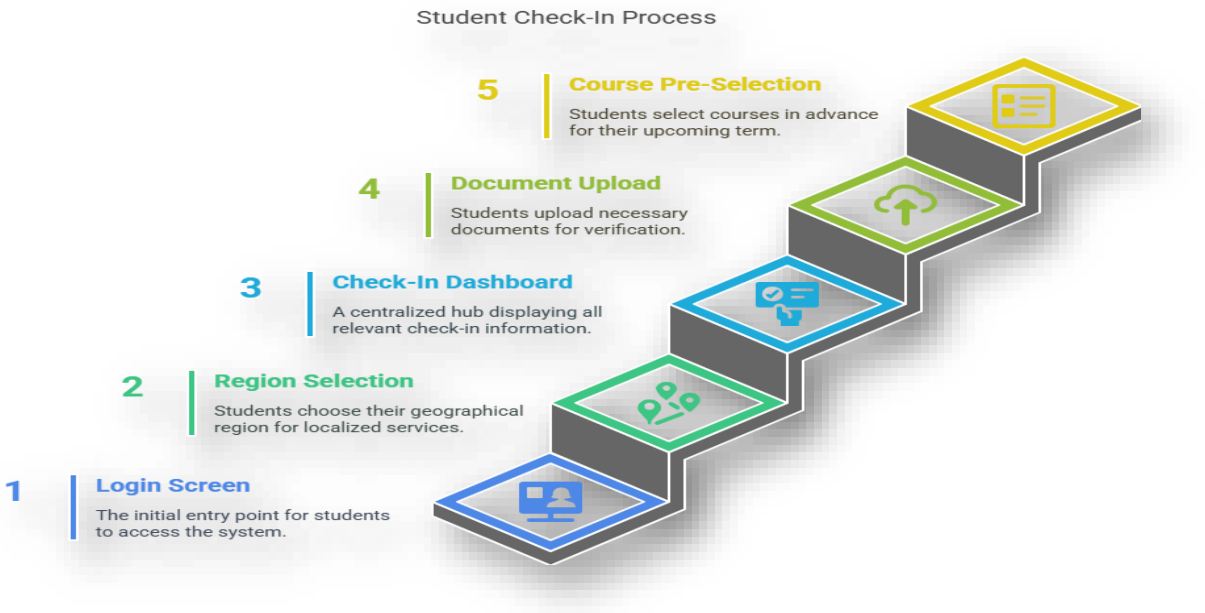

Figure 5: Workflow of the AI-Driven Student Check-In Process, illustrating the sequential stages of student onboarding, including login, region selection, dashboard access, document submission, and course pre-selection to support an efficient and user-centered check-in process.

Throughout these UI components, certain technical design patterns were employed. We used a controller class (CheckInAgentApp) to encapsulate all screens and shared data. This class approach (rather than a script of global functions) helps maintain state, such as the advisor_name which is stored as an instance variable once assigned. It also allowed creating helper methods like clear_screen to destroy old widgets when moving between screens, ensuring that each screen is drawn fresh without leftovers from the previous screen. The class initializes with the main Tk window and immediately builds the login screen, and later methods switch the context by building new screens. This structure is essential for a responsive GUI that can navigate between many steps.

Web Integration and Jupyter Compatibility:
An important aspect of our system's design is the ability to integrate the check-in agent into a web-accessible format for remote students. While Tkinter is fundamentally a desktop GUI toolkit, we considered the use of Jupyter Notebook as a means to launch the application on a server and allow remote access.

In practice, running a Tkinter interface through Jupyter can be non-trivial because Tkinter requires a display context. However, the concept of Jupyter compatibility here refers to using the Jupyter environment to host the logic and provide a web frontend. One approach is to use libraries like ipywidgets to create form elements in Jupyter that mimic our Tkinter UI, or to run a lightweight web server from the notebook (using frameworks like Flask) to serve the interface. Another approach is to use tools like BeeWare's Toga, which is a Python native GUI toolkit that can target web through

transpilation; but Toga is experimental for web frontends.

In our implementation, we primarily ensured that the code is modular and does not rely on system-specific calls, so it could be run in a headless mode or adapted to a web context. For instance, the logic for advisor assignment or document handling is separated from the UI event handling, so those parts can be invoked from a different interface. We also ensured that common Python packages are used (which are available in Jupyter environments, such as pillow and tkinter itself if the Jupyter server has GUI support).

This makes the project portable: educators or developers could open the notebook, run the application for demonstration, or step through parts of the code in an interactive manner. Additionally, the open-source nature of our solution means it can be hosted on platforms like GitHub and launched via Binder or similar, giving contributors and testers easy access. In summary, while the primary interface is desktop-based, the design is cognizant of web integration pathways, which is crucial for reaching students who may not be on campus to use a kiosk.

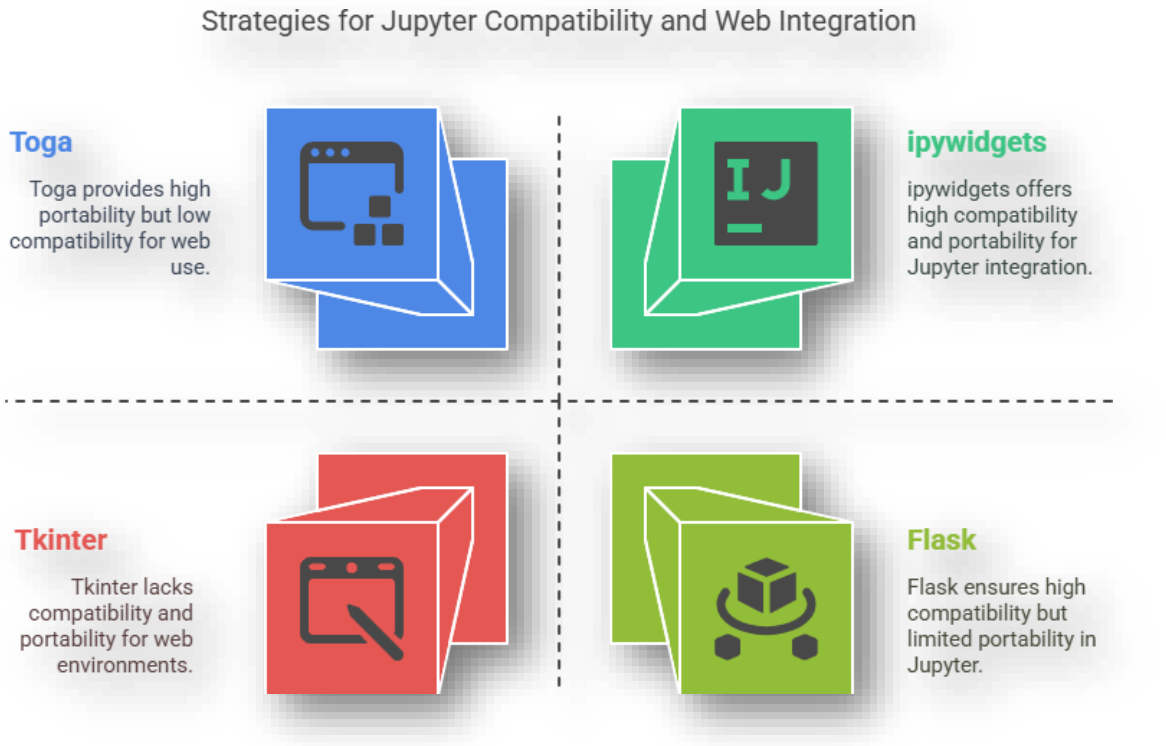

Figure 6: *Comparison of Toga, ipywidgets, Tkinter, and Flask based on their compatibility with Jupyter notebooks and web integration capabilities. The figure highlights the relative strengths and limitations of each framework, demonstrating that ipywidgets provides the most balanced solution for interactive Jupyter-based applications, while Flask is optimized for web deployment, Toga emphasizes portability, and Tkinter is primarily intended for desktop GUI development.*

JOURNAL OF
CLIMATE INNOVATION AND TECHNOLOGY (JCIT)
Advancing Research in Climate, Sustainability and Emerging Technologies
CLIMATE | SUSTAINABILITY | INNOVATION | TECHNOLOGY

Open-Source API Accessibility:

We emphasize that all components leverage open-source libraries or publicly accessible APIs to avoid reinventing the wheel. For example, rather than implementing image processing from scratch, we use Pillow (PIL) for any image resizing or format handling needed when loading logos or potentially processing student-uploaded images (like converting formats). For packaging and distribution, as mentioned, BeeWare's Briefcase and the Toga GUI toolkit can be used to deploy the agent on different platforms, which is preferable to proprietary solutions because it allows academic institutions to avoid platform lock-in. If we were to incorporate NLP for a chatbot extension, we could use open APIs such as the OpenAI GPT API or open-source models (provided usage complies with privacy policies), again maintaining flexibility and avoiding closed systems. By designing with open-source building blocks, the check-in agent can be maintained and extended by the community of developers at various institutions. It also fosters interoperability; for instance, if an institution uses a different authentication API or document storage service, those can be plugged into the system with minimal friction, given Python's extensive library support.

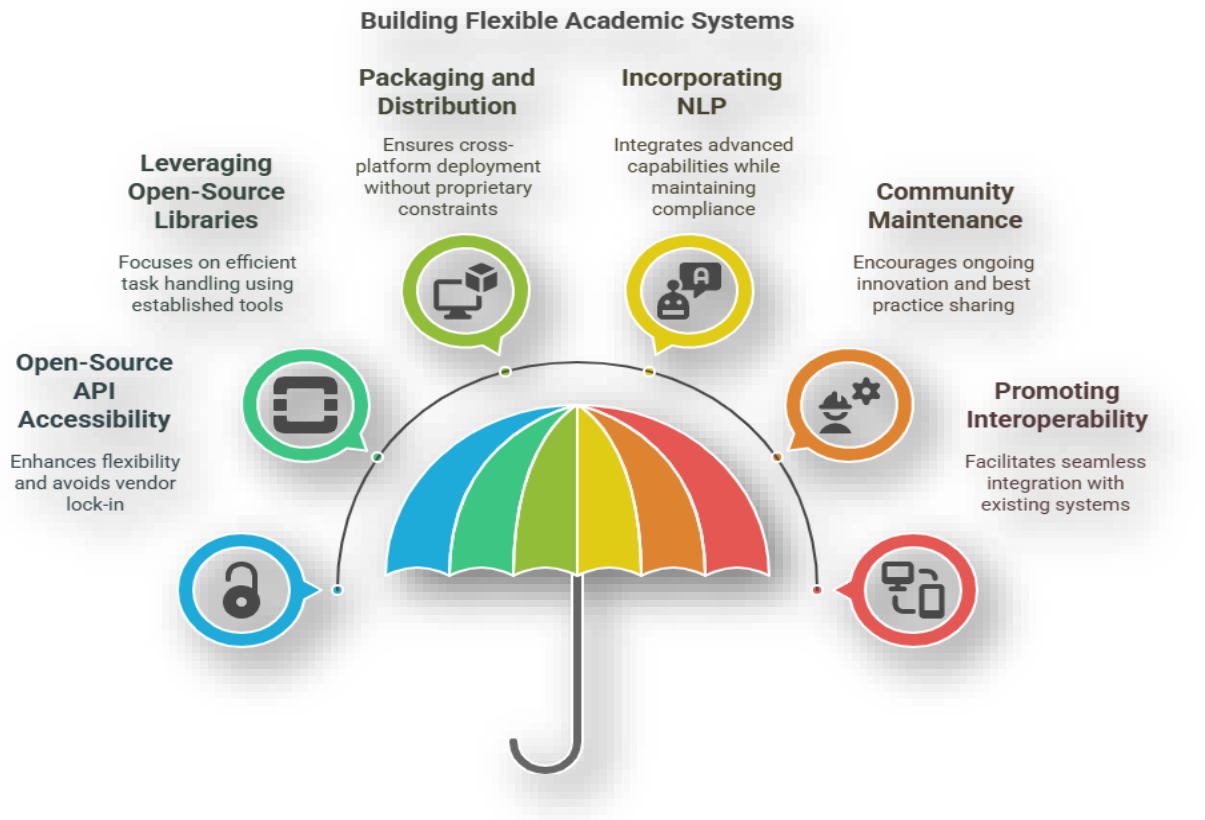

Figure 7: Core design principles for building flexible academic systems. The framework emphasizes open-source API accessibility, the use of open-source libraries, cross-platform packaging and distribution, NLP integration, community-driven maintenance, and interoperability to support scalable, maintainable, and adaptable academic software applications.

Implementation Details

In this section, we delve into the technical implementation of each major system component, providing insight into the code structure and algorithms used. The implementation was done in Python (v3.x) and primarily utilizes the Tkinter library for the GUI. Below, we break down the components: (1) Login & Authentication, (2) Advisor Assignment by Region, (3) Document Upload Module, (4) Course Pre-Selection Module, and (5) Status Tracking Dashboard. Each subsection describes the functionality and code approach, highlighting how the user-provided base code was generalized to an institutional-agnostic solution.

Login & Authentication: The login component is the entry point of the application. In our implementation, it consists of a simple GUI form that asks for a "Student ID" and a Login button. The code creates a Tkinter `Entry` widget for the student ID input and a `Button` widget for submission. When the button is clicked, it triggers a callback method (`region_selection_screen` in our case) after performing any necessary validation. In the provided base code, the login did not verify the ID against a database; it directly proceeded to the next step. For a generalized solution, this is where integration with an authentication system would occur. Depending on an institution's setup, this could involve: querying an internal API to verify the student's identity, using OAuth2 for single sign-on (if the application is integrated as a web app), or simply checking the format of the entered ID and ensuring it's not empty. The login screen's UI includes a header (logo and title) for clarity. We load a logo image via PIL if available (path can be configured), resizing it to fit the header. This is encapsulated in a conditional to handle cases where the image file might be missing – in which case a text placeholder is shown. This approach ensures the application is robust to missing resources. The design of the login component follows security best practices by not displaying any sensitive info – just a single input field. For enhanced security, the Student ID could be masked or supplemented by a password/PIN field if needed. However, often for check-in on a self-service kiosk, the student ID (or scanning a student QR code) is sufficient to identify the user, since the context is controlled.

Advisor Assignment (Region Selection):
After login, the next immediate task is assigning or confirming the student's advisor based on their geographic or program category. We implemented this with a region selection dropdown as described earlier. The code uses a Tkinter `OptionMenu` bound to a `StringVar` (`self.region_var`) that stores the chosen region. The list of regions is defined in a Python list, which makes it easy to modify without changing UI code. When the student submits their choice, the `display_advisor` function is executed. This function contains the logic to determine the advisor's name from the selected region. In the base code, a series of `if/elif` statements checks the region string and assigns an advisor's name accordingly. We retained this approach for clarity, though it could be improved by using a dictionary (`advisor_map = {"Nepal":"Shruti",...}`) for scalability, especially if dozens of regions are involved. Once the advisor is set (stored in `self.advisor_name`), a message box pops up to inform the user. The use of `messagebox.showinfo` is a straightforward way to give feedback; it creates a modal dialog that the student must acknowledge, ensuring they see the advisor assignment.

The agent then calls `check_in_progress()` to move to the next screen. If the student did not select a region (the default or an empty selection remains), the code triggers `messagebox.showwarning` to prompt them to choose a region. This prevents missing data. In an expanded implementation, the advisor assignment

could involve more complex logic: for instance, determining the student's country from their profile (if login was tied to the student database) and auto-selecting the region, or retrieving the advisor name from an external service. The current implementation is synchronous and on the client side (which is fine for a local app). If integrated with a web service, the app might send the student ID or region to an API endpoint to get the advisor info, which introduces asynchronous considerations. Python's simplicity allows either approach; for demonstration, local logic is used for determinism. Finally, a **Back** button on the region screen is provided using a helper (create_back_button) that simply calls the login screen creation method. This gives users the flexibility to go back and change their ID if needed (or if they accidentally proceeded with the wrong account).

Document Upload Module: The document upload functionality primarily revolves around allowing the user to select a file and then handling that file. With Tkinter, we utilize the file dialog as mentioned. The code snippet responsible is the upload_document method, which calls filedialog.askopenfilename(). This function opens the OS-specific file chooser and returns the path of the selected file. The method then checks if a file was indeed selected (the dialog returns an empty string if cancelled). If a path is present, we show a confirmation message box with the name of the file. In the generalized version, instead of (or in addition to) showing a message, we would send the file to a server.

This can be done with Python's requestslibrary to POST the file to a REST API, or using an SDK if the files are to be stored in cloud storage (for example, the Azure Storage SDK or AWS S3 boto3 client). It's important that such upload happens over HTTPS and that the file is stored securely (with access control so only authorized staff can view it). In our design, after an upload, the system could update an internal status variable (e.g., self.docs_uploaded = True) and when returning to the status dashboard, "Document Status" could change from "Pending" to "Uploaded" or "Under Review". We also included a "Back" button on the upload screen to return without uploading, in case the student clicked it by mistake or needs to gather their files. One consideration is ensuring the application can handle multiple document uploads.

The current UI handles one at a time (the user can click "Upload Document" again if multiple files are needed, each click can be a separate upload event). A future refinement might be to list required documents explicitly and allow an upload for each, ensuring completeness. Nevertheless, the implemented module confirms that using Tkinter's dialogs and message boxes can create a smooth user experience for file submission in a desktop setting.

Course Pre-Selection Module: The course pre-selection screen's implementation uses similar patterns to the region selection. A StringVar and OptionMenu present the available courses. The default selection is set to the first course in the list. The "Submit Selection" button is wired to a lambda function that simply shows a success info dialog. This lambda could be replaced with a more complex function that, for example, saves the choice to a database or calls an API to reserve a seat in the chosen class. Since the UI currently only allows picking one course at a time, a student would have to reopen the screen to pick another; an iterative selection process would need more feedback (like adding the selected course to a list and perhaps showing remaining slots).

In an academic context, course pre-selection might be constrained by available slots or prerequisites, but our agent assumes a simplified scenario where any chosen course is fine. Implementation-wise, the module demonstrates how easy it is to incorporate a dropdown and a button to gather user input in Tkinter. The actual list course_options can be dynamically generated. For instance, when the student logs in, the system could determine their intended major (from an API or a data file) and then populate course_options with recommended first-semester courses for that major. Python makes such data handling straightforward (just populate the list before creating the OptionMenu). One technical note is that we did not store the course selection persistently in the prototype – it's a transient action that only triggers a message. For real use, maintaining an internal state (e.g., self.selected_course) or writing to a file would be necessary to remember the choice. Also, after submission, one might disable the selection or mark it as completed on the status screen to prevent duplication. Our implementation leaves these as logical extensions, focusing on the core interaction loop.

Status Tracking Dashboard: The check-in progress screen, as described, is a composite view that reflects what steps are done or pending. In code, this is a series of Label widgets and Button widgets assembled vertically (using Tkinter's pack layout manager with padding). The labels show static text combined with dynamic content (for advisor name, we use an f-string to include self.advisor_name in the label text when creating it). This dynamic insertion means the label reflects the actual advisor assigned earlier. If no advisor were assigned (somehow, if the user jumped to this screen out of sequence), that variable would be empty – another reason the workflow is designed to enforce order. The status screen's buttons for "Upload Documents" and "Pre-Select Courses" are each tied to their respective screen functions. So, clicking those essentially navigates into a sub-process and returns back when done. One could think of the status screen as a main menu for the remaining tasks. The implementation ensures a Back button is also present to allow retreating to the previous stage (in this case, we allowed going back to region selection if needed, via create_back_button(self.region_selection_screen)).

In a deployed scenario, going back to region selection might be disabled after some point (since switching advisor after uploading documents might complicate the workflow), but we included it to reflect flexibility during development and testing. Moreover, having a back navigation is useful if a student mis-selected a region and immediately realizes it – they can correct it. The bottom of the status screen (like all screens in our design) includes a footer with the institution's copyright notice. This is cosmetic but part of a professional application's completeness. It also subtly indicates to the user that the application is officially sanctioned (since it carries the university name and year).

From a coding perspective, the transitions between screens (login -> region -> status -> upload or course and back) are managed by calling the respective screen creation methods. Each of those methods internally calls clear_screen at the start to remove old widgets, then populates new ones. This simplistic approach is effective, though in more complex GUIs one might use a stacked frame approach (multiple frames and raise/lower them instead of destroying and recreating every time). We chose recreation to ensure that each time you enter a screen, it reflects the latest state (for example, if we extended the status screen to dynamically check a database for "Verified" status on documents, recreating it would fetch the new status each time). The performance overhead for a small number of widgets is negligible.

Throughout the implementation, we maintained a focus on clarity and user guidance. Each screen has

a clear title label indicating what the student should do (e.g., "Document Upload", "Course Pre-Selection", etc. in bold, slightly larger font). Consistent color styling is applied: in the updated code, we switched to a dark blue background with yellow/gold text for a higher contrast, representing a more professional theme consistent with some university colors. Buttons are styled with a solid background color and white text, using slight variations (teal vs dark blue) for primary vs secondary actions in the earlier version, and a more uniform style in the updated version with a single accent color for all action buttons. These choices can be easily changed in the code by modifying the color codes in one place (we defined them inline for demonstration, but they could be constants at the class level).

Finally, we ensured that our code is compatible with being run as a script or module by encapsulating the launch in the `if __name__ == "__main__":` block. This means the application can be started by running the Python file directly, which is how a standalone app would work, or it can be imported and the class used in other contexts (for instance, to be integrated with another interface). This is a minor software engineering point, but important for generalizing the solution beyond the initial prototype.

Security and Data Privacy Considerations

Designing an AI-driven check-in agent requires careful attention to security and privacy, given the sensitive personal data involved. In an educational context, student records and identification documents are protected by various laws and regulations. In the United States, the Family Educational Rights and Privacy Act (FERPA) mandates strict controls on education records and limits disclosure of personally identifiable information without student consent. Our system, which handles student IDs, academic and immigration documents, and potentially enrollment information, must ensure FERPA compliance.

This involves both technical and policy measures: authentication should ensure that only the student (or authorized staff) can access their records, data in transit (such as uploaded documents) should be encrypted (e.g., using HTTPS and secure file transfer to the server), and data at rest should be stored securely with access controls. For instance, if the agent uploads a passport scan to cloud storage, that storage bucket must have restricted permissions so that only the international office staff (and not just any authenticated university user) can see it. Moreover, the system should log access to sensitive data in case of audits and to provide transparency if a student requests an account of who accessed their records (a FERPA right).

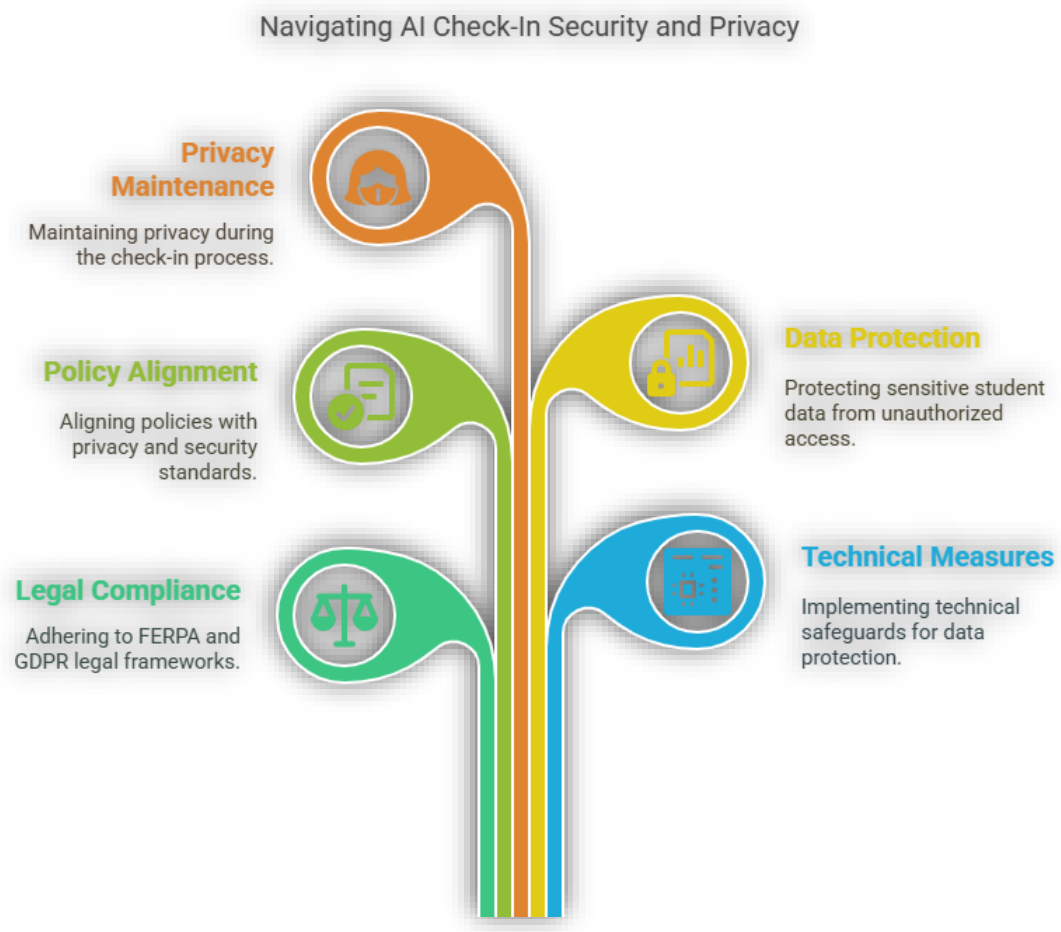

Figure 8: Security and privacy framework for AI-based student check-in process. The framework identifies five essential dimensions—privacy maintenance, policy alignment, legal compliance, data protection, and technical security measures—that collectively support the secure, ethical, and compliant implementation of AI technologies in higher education.

For international students, we also consider the impact of global data protection regulations. If the system serves students who are abroad or if it's hosted in a way that interacts with data centers globally, the EU's General Data Protection Regulation (GDPR) might be relevant. GDPR imposes requirements such as obtaining clear consent for data collection, allowing users to request deletion of their data, and appointing a Data Protection Officer if large-scale processing is involved. While our check-in agent is intended for university use (thus typically falling under educational purposes, with the student as a willing participant in providing information), it should still have clear privacy

notices. When a student first uses the system, the university could present a disclaimer or consent form explaining what data will be collected (e.g., "This system will store your uploaded documents and share them with the International Student Services office for the purpose of completing your check-in. Your data will be protected under university privacy policies and applicable law.").

In the implementation, certain precautions were already taken at the design level. For example, we do not store the student's password or any highly sensitive secret in the application; authentication is expected to be handled by external secure systems. The agent also doesn't locally cache the documents – it would pass them to a server which can be secured and audited, rather than keeping files on the kiosk device. This is important because kiosk devices could be stolen or compromised; they should ideally not hold any data once the session ends. For added security, the application should implement session timeouts or log-out functionality. If a student walks away after logging in, the system should auto-logout after a short period to prevent others from hijacking the session. Our prototype didn't implement this, but the framework allows adding a timer reset on each interaction and a timer expiration that calls a logout routine (which could just send the app back to the login screen and clear any cached state).

Another security aspect is input validation. We have simple validation like requiring region selection, but a robust system should validate all user inputs. For instance, ensure the student ID entered matches expected patterns to prevent SQL injection or other code injection if the ID is used in queries. Since our system is in Python and uses parameterized data (not direct SQL in the client), the risk on the client side is low. But if hooking into an API, always sanitize or use safe query methods on the server. The file upload should also be scanned – since students might inadvertently or maliciously upload a file with a virus. The system should not execute any code from uploaded files and should ideally run virus scanning in the backend. Many universities use antivirus on file upload or restrict types (e.g., only PDF, JPEG images, etc., disallowing executables).

Data encryption is vital for privacy. If the check-in agent stores any data locally (even temporarily), it should be encrypted. For example, if the app writes a temporary file with the student's info or downloads a template, using encryption or at least secure file permissions is advisable. In our current design, most data is transient (in-memory variables or shown on screen). The network communication (if any) would rely on external APIs' security (so choosing HTTPS endpoints, etc.). We also consider that the AI component might involve external services (like a cloud NLP service). If we integrate something like Dialogflow or Watson Assistant for answering questions, any data sent to those services (like a student's question about their status) could potentially include personal info. We should configure such services to not store conversation data, or use self-hosted AI models to keep data on university servers. Many cloud AI providers allow opting out of data logging for privacy, and that would be essential here.

From a policy perspective, the system should align with institutional data handling policies. Many universities have an "acceptable use" policy for student data and specific guidelines for services interacting with student records. Our agent would likely be categorized under tools that manage student records, so it would need approval from the university's data governance or IT security office. Ensuring compliance with standards like SOC2, ISO 27001 (if the system is cloud-hosted) might also be relevant when scaling up.

One unique aspect of data privacy for international students is the handling of immigration data. Documents like visas and passports are highly sensitive. The system should perhaps avoid displaying full passport numbers or visa numbers on the screen after upload (to minimize shoulder-surfing risks in a kiosk environment). It might show a masked confirmation like "Passport scan uploaded" without exposing details. Similarly, once an advisor is assigned, the student should see their name, but maybe not the advisor's detailed contact info unless necessary, to avoid any inadvertent privacy issues for the advisor.

In sum, our implementation was done with a security-conscious mindset (using minimal local storage, leveraging existing secure services for heavy lifting, validating inputs). For an actual deployment, further hardening is needed, but the design is compatible with those enhancements. Educational institutions can integrate this agent into their secure infrastructure, benefiting from the streamlined check-in while still upholding the privacy rights and protections owed to students.

Challenges and Future Improvements

Developing the AI-driven check-in agent revealed several challenges and areas for potential improvement. Some challenges were technical, such as ensuring a seamless user experience across different platforms, while others were organizational, such as encouraging user adoption and maintaining the system's content. This section discusses these issues and outlines future enhancements, including machine learning-driven personalization, multilingual support, and cloud integration, to evolve the system into a more intelligent and robust tool.

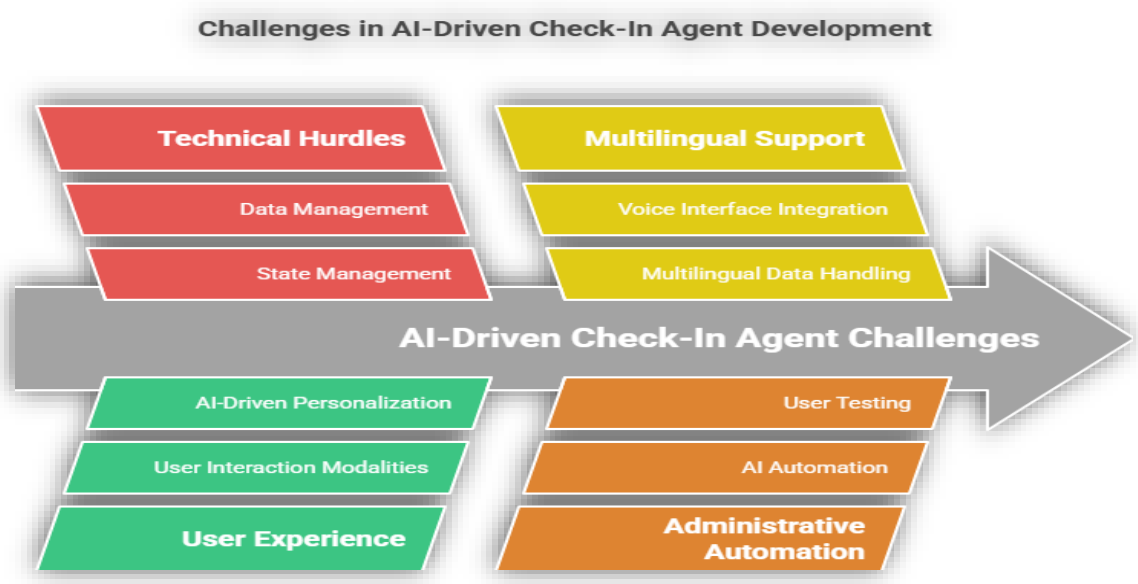

Figure 9: Major challenges in the development of AI-driven student check-in agents. The framework categorizes challenges into technical hurdles, multilingual support, user experience, and administrative automation, emphasizing the technical, operational, and user-centered factors that influence the successful deployment of AI-based check-in solutions in higher education.

Challenges: One immediate challenge is **user interaction modality**. Our current system uses a GUI with buttons and drop-down menus, which is straightforward but not conversational. If users expect a chatbot-like experience (e.g., to type or speak questions), the system might feel rigid. Conversely, if they expect a form-driven portal, the presence of any conversational agent might confuse them. Striking the right balance and perhaps offering both options (guided workflow and a chatbot helper) is a design challenge. We have planned for chatbot integration, but implementing it requires careful UI/UX design to not overwhelm the user with too many options.

Another challenge is **state management and concurrency**. In a scenario where multiple students might use the system (say multiple kiosks or a web version with many users at once), ensuring each session is isolated and that back-end updates (like document verification by staff) reflect in the student's view requires a more complex architecture (possibly client-server with real-time communication). Our standalone prototype doesn't handle concurrent sessions since it's single-user. Scaling up might involve converting the app to a web application where the server keeps track of sessions (through tokens or login sessions) and the client (browser or app) requests status updates. Implementing real-time updates (for example, if an advisor marks a document as approved, the student's status screen could update automatically) might involve WebSocket connections or periodic polling. These additions complicate the implementation but would significantly enhance the user experience.

Data management poses another challenge. Currently, things like region-to-advisor mappings and course lists are hard-coded. In reality, these data change – advisors may go on leave, new courses are added, etc. Maintaining the system requires an easy way to update such information, ideally through an admin interface or by reading from a master data source. If not carefully managed, the system could provide outdated guidance (e.g., assigning a student to an advisor who is no longer available). A future improvement is to connect these components to live data sources: e.g., integrate with the university's HR system or advisor assignment list for up-to-date advisor info, and integrate with the course catalog for available classes each term. This highlights that the check-in agent would benefit from **cloud integration** – hosting parts of the solution on a cloud platform could allow it to query current data and even use cloud functions to trigger notifications (like sending an email to the advisor when a student completes check-in).

From the perspective of AI and personalization, our current agent is largely rule-based. A major future enhancement is to incorporate **machine learning-driven personalization**. For instance, the system could learn from past students' behaviors to recommend actions. If it notices that students from a particular region often miss a certain form, it could proactively emphasize that step ("Students from your region often require form X, please ensure you have submitted it"). Personalization could also apply to course recommendations: using the student's academic background (perhaps their prior coursework or stated interests), the system could suggest courses rather than presenting a generic list. Implementing this would require gathering data on student choices and outcomes, training recommendation models, and validating them for fairness and accuracy. Caution is needed to avoid bias – the recommendations should not inadvertently track irrelevant features (for example, recommending courses based on region might be inappropriate unless it's actually correlated with program prerequisites).

Multilingual support is another important improvement, particularly for international student services. While English is the primary language of instruction in many U.S. institutions, providing check-in instructions in a student's native language can greatly enhance comfort and reduce errors. We plan to internationalize the interface by externalizing all text strings and providing translations for common languages (e.g., Spanish, Chinese, Arabic, French). Python has libraries like gettext for managing translations. Additionally, an AI chatbot could detect the language of a user's question and respond accordingly, possibly by leveraging translation APIs or a multilingual language model. However, multilingual support brings challenges in ensuring the accuracy of translations (especially for technical terms like "transcript" or "immunization record") and maintaining the translations as the English source text changes. It might also require cultural adaptation – e.g., different date formats, name order, etc., if those are presented.

We also encountered a challenge in testing the system with real users. As a future step, we intend to conduct user testing sessions with a small group of international students to gather feedback on usability. This can highlight unforeseen issues, such as parts of the interface that are confusing or steps where students hesitate. Their feedback might suggest additional features, like a progress indicator (e.g., 3 of 5 steps completed) or a summary page before final submission. It could also show where our assumptions might not hold – for example, maybe students want to upload documents from their phone camera directly (which suggests having a mobile app or at least making the web version mobile-friendly).

On the technical side, integrating a voice interface could be a future enhancement. Some kiosks or devices might allow spoken interaction, which could be useful for students who have difficulty typing in English. Libraries like Speech Recognition for Python or cloud services could enable the student to speak their student ID or ask questions verbally. Combined with a text-to-speech output, the agent could essentially "talk" the student through the process. This multimodal interaction is ambitious but aligns with trends in user experience for AI assistants.

Finally, the use of AI can be extended to the administrative side of the check-in process. For example, using computer vision to auto-classify or extract information from uploaded documents (passport OCR to get the name and verify identity, or transcript parsing to check if certain requirements are met). Our current system doesn't do this, but as a future improvement, we could integrate an OCR (Optical Character Recognition) engine or a machine learning model trained on document recognition. This would transform the agent from just a passive receiver of files to an active verifier that can immediately alert a student if a document is illegible or if they uploaded the wrong file. It also provides a layer of AI assistance to the staff by pre-validating submissions (e.g., "the visa expiration date is missing on the scan, please upload a clearer image").

Future Work Summary: To summarize, future iterations of the check-in agent will focus on: (1) **Conversational AI integration** – blending the GUI workflow with a chatbot that can answer questions and accept inputs in natural language; (2) **Cloud-based architecture** – enabling multi-user support, real-time updates, and connection to live data sources and services (including cloud storage, databases, and possibly serverless functions for processing); (3) **Personalization and ML** – using student data and behavior analytics to tailor the experience and provide smart recommendations or alerts; (4) **Internationalization** – supporting multiple languages and

accommodating cultural differences in the check-in process; (5) **Enhanced UI/UX** – refining the interface with progress tracking, help tooltips, and possibly voice interaction; and (6) **Automated document processing** – employing AI to assist in verifying and extracting data from submitted documents.

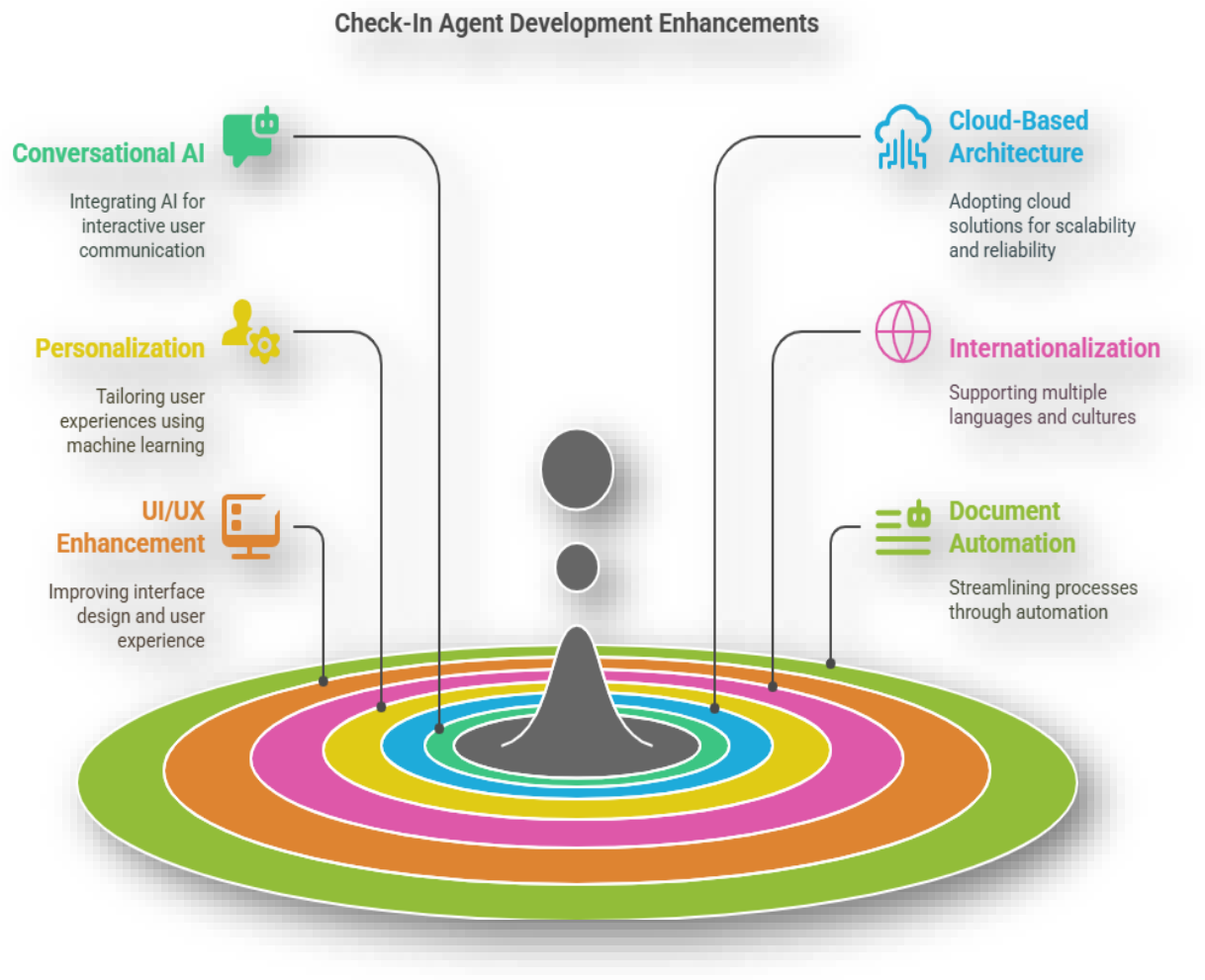

Figure10: enhancements for AI-driven student check-in agent development. The framework highlights six strategic enhancement areas—conversational AI, personalization, UI/UX enhancement, cloud-based architecture, internationalization, and document automation—that collectively improve system intelligence, scalability, accessibility, and administrative efficiency in higher education environments.

Each of these improvements will require careful evaluation to ensure they truly benefit the end-users

(students and staff) and do not inadvertently introduce new issues (such as confusing the users or risking privacy). Nonetheless, these directions hold promise for making the check-in agent a comprehensive digital assistant in the international student enrollment journey.

Conclusion

The AI-driven check-in agent presented in this paper offers a novel approach to streamlining the onboarding of international and transfer students. By combining a structured, user-friendly GUI with the potential of AI enhancements, the system addresses common bottlenecks in the check-in process – from lengthy queues for document verification to confusion about course registration and advisor assignment.

The generalized design means that any institution can adapt the framework to its own workflows, replacing or extending modules as needed (for example, plugging in their list of regional advisors or linking to a specific student information system). Our literature review highlighted how both U.S. and global institutions are increasingly leveraging chatbots and AI for student services; this agent builds on that trend by integrating those capabilities into a holistic check-in solution rather than a standalone Q&A chatbot.

From a technical standpoint, the implementation demonstrates that accessible technologies (Python, Tkinter, and open libraries) are sufficient to create a functional and flexible application for student check-ins. The use of open-source tools like BeeWare further ensures that even resource-constrained institutions can deploy the agent to various platforms (desktops, web, or mobile) without significant cost. This open approach encourages collaboration – universities can share improvements or tailor the agent to emerging needs (such as adding health screening questions, which became relevant during the COVID-19 pandemic, or connecting the agent to new systems as their IT ecosystem evolves).

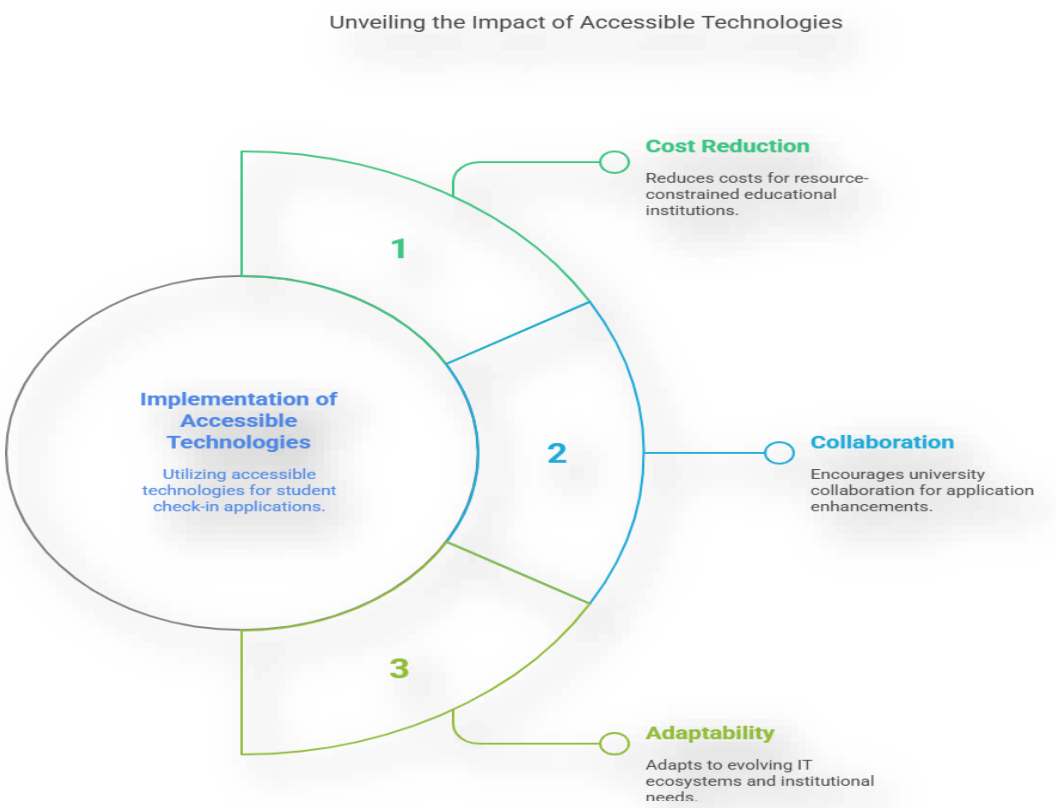

Figure 11: This figure illustrates the implementation of accessible technologies for student check-in applications. It highlights three key benefits: **cost reduction**, **collaboration**, and **adaptability**, which collectively enhance institutional efficiency, foster inter-university cooperation, and support evolving educational technology needs.

Institutional impact of such an AI-driven agent can be significant. First, there are **efficiency gains**: administrative staff in international student offices can offload routine tasks (like initial document collection and explaining standard procedures) to the agent, freeing them to focus on complex cases and one-on-one interactions that truly require human expertise. This not only improves productivity

but can also reduce errors, as the standardized process ensures every student submits all required information. Second, the agent improves **process transparency and student empowerment**.

Students receive immediate feedback on their check-in status and know exactly which steps to complete, which aligns with best practices of process clarity. Real-time notifications (which can be added via email/SMS integration) keep students informed, preventing them from missing crucial requirements. In turn, this leads to faster completion of check-in and fewer last-minute issues when classes begin.

User satisfaction is another expected benefit. Modern students are digital natives who often prefer self-service options for administrative tasks. An AI-driven check-in agent provides a modern, interactive experience that can reduce the anxiety of navigating a new environment. Especially for international students who may arrive on campus just before semester start, being able to complete many onboarding steps online or at a kiosk at their convenience is a substantial relief. The personalized elements – like being immediately told who their advisor is and potentially receiving tailored course suggestions – make the experience feel customized and welcoming, rather than bureaucratic. Over time, this can contribute to a better first impression of the institution and improve overall student satisfaction scores related to the onboarding process.

Furthermore, the data gathered by the check-in agent can help institutions in a continuous improvement loop. By analyzing which questions students frequently ask the chatbot or which documents are often uploaded incorrectly, the university can identify pain points in communications or adjust their orientation programs. It also provides an aggregated view of progress during the intake period (e.g., how many students have completed all steps), allowing administrators to allocate resources dynamically (if many students are stuck on a step, perhaps more support or clarification is needed).

In conclusion, the AI-driven check-in agent exemplifies how artificial intelligence and intuitive software design can transform administrative processes in education. It stands on the shoulders of prior chatbot solutions and process management systems, unifying their strengths into a single application geared toward a critical phase of the student lifecycle. While our implementation is a prototype, it serves as a reference architecture that is ready to be expanded and deployed in the real world. As future enhancements are added – from multilingual chat capabilities to deeper integration with institutional systems – the agent can grow in its role, potentially becoming a comprehensive digital assistant that accompanies students from pre-arrival to graduation. The work reported in this paper is an important step in that direction, and we anticipate that it will spur further innovation and collaboration among universities seeking to leverage AI for student success.

References

[1] A. Alaqoyli and H. Abdelhafez, "Intelligent Chatbot for Admission in Higher Education," International Journal of Information and Education Technology, vol. 13, no. 9, pp. 1348–1357, 2023.

[2] T. T. Nguyen, A. D. Le, H. T. Hoang, and T. Nguyen, "NEU-Chatbot: Chatbot for Admission of National Economics University," Computers & Education: Artificial Intelligence, vol. 2, p. 100036, 2021.

[3] L. C. Page and H. Gehlbach, "Freezing 'Summer Melt' in Its Tracks: Increasing College Enrollment with AI," Brookings Institution, 2017.

[4] P. Smutny and P. Schreiberova, "Chatbots for Learning: A Review of Educational Chatbots for the Facebook Messenger Platform," Computers & Education, vol. 151, p. 103862, 2020.

[5] U.S. Department of Education, "Family Educational Rights and Privacy Act (FERPA)," Washington, DC, USA, 2021.

[6] Terra Dotta, "International Student & Scholar Services Software Features," 2023.

[7] BeeWare Project, "BeeWare—Write Python. Run Everywhere," Available: https://beeware.org

[8] A. Al-Abdullatif, A. Al-Dokhny, and A. Drwish, "Implementing the Bashayer Chatbot in Saudi Higher Education," Frontiers in Psychology, vol. 14, 2023.

[9] B. W. Bimpong, "The Impact of Generative AI Educational Chatbots on Academic Support Experiences of Students in U.S. Research Universities," NASPA Blog, 2025.

[10] A. M. Cox, S. Pinfield, and S. Rutter, "The Intelligent Library: Thought Leaders' Views on the Likely Impact of Artificial Intelligence on Academic Libraries," Library Hi Tech News, vol. 40, no. 4, pp. 26–29, 2023.

[11] B. Hurter, "The Role of AI Chatbots for Higher Education Success in 2024," Element451 Blog, 2024.

[12] C. W. Okonkwo and A. Ade-Ibijola, "Chatbots Applications in Education: A Systematic Review," Computers & Education: Artificial Intelligence, vol. 2, p. 100033, 2021.

[13] L. C. Page and H. Gehlbach, "How an Artificially Intelligent Virtual Assistant Helps Students

Navigate the Road to College," AERA Open, vol. 3, no. 3, pp. 1–12, 2017.

[14] A. Winkler and M. Söllner, "Unleashing the Potential of Chatbots in Education: A State-of-the-Art Analysis," Academy of Management Annual Meeting Proceedings, 2018.

[15] S. K. Adamopoulou and E. Moussiades, "An Overview of Chatbot Technology," IFIP Advances in Information and Communication Technology, vol. 584, pp. 373–383, 2020.

[16] M. A. Alam, "Artificial Intelligence in Higher Education: Opportunities and Challenges," Education and Information Technologies, vol. 29, pp. 1099–1123, 2024.

[17] UNESCO, "Guidance for Generative AI in Education and Research," Paris, France, UNESCO Publishing, 2023.

[18] OECD, "Artificial Intelligence in Education: Challenges and Opportunities for Sustainable Development," Paris, France, OECD Publishing, 2021.

[19] European Union, "General Data Protection Regulation (GDPR)," Regulation (EU) 2016/679, Brussels, Belgium.

[20] OpenAI, "GPT-4 Technical Report," arXiv:2303.08774, 2023.

[21] T. B. Brown et al., "Language Models are Few-Shot Learners," Advances in Neural Information Processing Systems (NeurIPS), vol. 33, pp. 1877–1901, 2020.

[22] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," Proc. NAACL-HLT, pp. 4171–4186, 2019.

[23] A. Vaswani et al., "Attention Is All You Need," Advances in Neural Information Processing Systems (NeurIPS), vol. 30, 2017.

[24] T. Chen, M. Guestrin, et al., "XGBoost: A Scalable Tree Boosting System," Proc. ACM SIGKDD, pp. 785–794, 2016.

[25] Rasa Technologies, "Rasa Open Source Conversational AI Framework," Available: https://rasa.com

[26] Google, "Dialogflow CX Documentation," Google Cloud Platform, 2024.

[27] Microsoft, "Azure AI Bot Service Documentation," Microsoft Learn, 2024.

[28] Python Software Foundation, "Tkinter—Python Interface to Tcl/Tk," Python Documentation, 2024.

[29] Pillow Developers, "Pillow (Python Imaging Library Fork)," Available: https://python-pillow.org

[30] J. M. Wing, "Computational Thinking," Communications of the ACM, vol. 49, no. 3, pp. 33–35, 2006.