Scholars Journal of Engineering and Technology
OPEN ACCESS
Abbreviated Key Title: Sch J Eng Tech
ISSN 2347-9523 (Print) | ISSN 2321-435X (Online)
Journal homepage: https://saspublishers.com

AI-Driven Maui Framework App Development and SDK Test Validation for Fintech Clients

Jeshwanth Ravi[1*]

[1]Software Test Engineer – Sr. Consultant, Visa Inc, Austin Texas, US

DOI: https://doi.org/10.36347/sjet.2025.v13i06.001 | **Received:** 31.04.2025 | **Accepted:** 04.06.2025 | **Published:** 11.06.2025

***Corresponding author:** Jeshwanth Ravi
Software Test Engineer – Sr. Consultant, Visa Inc, Austin Texas, US

Abstract **Review Article**

This research investigates the efficacy and challenges of employing generative Artificial Intelligence (AI) models—specifically Anthropic Claude, OpenAI GPT, and Google Gemini, orchestrated via the Cline extension in Visual Studio Code—to construct a .NET MAUI test application for validating FinTech Software Development Kits (SDKs). The study focuses on automating the generation of XAML for user interfaces and C# for backend logic, targeting critical FinTech workflows such as wallet provisioning on Android and iOS platforms. The methodology involved an iterative AI-assisted development process, encompassing AI-driven planning, code generation, extensive human-led refinement, and rigorous testing using a mock SDK and Appium for UI automation. Hypothesized results suggested significant acceleration in initial boilerplate code generation, though a substantial portion (40-60% for XAML, 30-50% for C#) required manual rework to address framework-specific nuances, ensure code quality, and implement robust error handling. Key challenges identified include the AI's inconsistent understanding of .NET MAUI's XAML dialect, limitations in managing complex UI state, occasional AI hallucinations, and the need for highly specific, context-rich prompts. Despite these hurdles, the final human-refined test application successfully automated the validation of the designated FinTech workflow across both platforms. The findings indicate that while AI serves as a powerful accelerator in test application development, expert human oversight remains indispensable for ensuring the quality, security, and framework compliance of the generated code. The study concludes that AI dramatically reshapes the role of the test automation engineer towards that of an AI orchestrator and critical validator, and outlines future research directions including the development of fine-tuned AI models for specific frameworks like .NET MAUI and AI-driven test data generation for complex FinTech scenarios.

Keywords: NET MAUI, FinTech SDK, Test Automation, AI Code Generation, Anthropic Claude, OpenAI, Google Gemini, Cline, Wallet Provisioning, Mobile Application Testing.

INTRODUCTION

The domain of cross-platform mobile application development has witnessed a significant transformation with the advent of frameworks aiming to unify codebase management while delivering native performance and user experiences. NET Multi-platform App UI (MAUI) represents Microsoft's latest advancement in this arena, evolving from Xamarin. Forms to offer a more streamlined development paradigm. [1]. NET MAUI facilitates the creation of applications for Android, iOS, macOS, and Windows from a single C# codebase. Its architecture is engineered to provide a unified project structure, which simplifies dependency management and platform-specific code organization compared to its predecessors [1]. Furthermore, NET MAUI introduces an enhanced UI consistency mechanism through the use of "handlers," which provide a more flexible and performant way to map cross-platform UI controls to native platform controls, and it promises improved overall application performance [1]. These characteristics position.NET MAUI as a compelling choice for developing FinTech applications, which often demand robust native capabilities, a consistent user experience across multiple platforms, and the ability to handle potentially high-load enterprise scenarios. The integration with the broader. NET ecosystem and the availability of.NET developers further contribute to its suitability for long-term FinTech projects.

The architectural simplification offered by.NET MAUI, particularly its single project structure, directly addresses historical complexities associated with managing multiple platform-specific projects, a common

pain point in Xamarin development [1]. This streamlined approach can lead to more efficient code management and reduced development overhead. However, this architectural improvement in code organization does not inherently diminish the complexities involved in testing interactions with platform-specific SDKs. FinTech applications, in particular, frequently rely on SDKs that interface directly with native operating system features such as secure storage, biometric authentication, and specialized hardware communication. While.NET MAUI abstracts many platform differences, the underlying native interactions facilitated by these SDKs still necessitate rigorous, platform-aware testing strategies to ensure functionality, security, and compliance. Thus, while MAUI's architecture simplifies code management, the burden of validating these critical native integrations remains substantial, underscoring the need for effective and comprehensive testing methodologies.

FinTech SDKs are instrumental in the modern financial services landscape, providing the building blocks for a wide array of functionalities including, but not limited to, payment processing, direct banking integration, robust identity verification (KYC/AML), and mobile wallet provisioning [6]. These SDKs empower FinTech companies, traditional financial institutions, and other businesses to rapidly incorporate sophisticated financial services into their applications, thereby accelerating innovation and time-to-market [8]. The adoption of well-engineered SDKs can significantly accelerate customer integration timelines, enhance the developer experience by abstracting underlying complexities, ensure the reliability of API connections, bolster application security through pre-vetted components, and simplify the process of managing updates and evolving functionalities [8].

Given the critical nature of the services they enable, the thorough validation of FinTech SDKs is of paramount importance. Financial transactions are inherently high-value and sensitive, demanding the utmost accuracy and security. The FinTech industry is also subject to stringent regulatory frameworks, such as the General Data Protection Regulation (GDPR) and the Payment Card Industry Data Security Standard (PCI DSS), which impose strict requirements on data handling and security [10]. Furthermore, customer trust is a cornerstone of any financial service; any lapse in security or functionality can lead to severe consequences, including direct financial losses, regulatory penalties, and irreparable reputational damage. The increasing complexity arises from the fact that modern FinTech services are often a tapestry of interconnected components, frequently delivered via multiple specialized SDKs. Each SDK, while providing essential functionality, also introduces a potential point of failure or vulnerability if not properly integrated and validated.[7] Some SDKs may also present a "black box" testing challenge, where internal workings are obscured, making comprehensive validation difficult [12]. The task of validating not only the SDK in isolation but also its seamless and secure integration within a mobile application, across diverse platforms like Android and iOS [12], becomes exponentially more intricate. This escalating complexity underscores a critical need for more efficient, scalable, and intelligent testing approaches, moving beyond traditional manual methods which can become significant bottlenecks in rapid development cycles.

The emergence of Artificial Intelligence (AI), particularly generative AI (GenAI) powered by Large Language Models (LLMs), is beginning to reshape the software development landscape. Prominent models such as Anthropic's Claude series [15], OpenAI's GPT family [17], and Google's Gemini are increasingly being employed to automate and augment various stages of the Software Development Lifecycle (SDLC) [20]. In the realm of code generation, AI tools demonstrate the capability to produce boilerplate code, complete functions, and even user interface (UI) components in languages like C# and markup languages like XAML [15]. Integrated Development Environment (IDE) extensions, such as Cline for Visual Studio Code, are facilitating this by embedding these powerful AI models directly into the developer's workflow [22]. Beyond code generation, AI is also making inroads into software testing, with applications in automated test case generation from requirements or user stories, creation of realistic test data, development of self-healing test scripts that adapt to UI changes, and predictive bug analysis based on historical data [20].

The increasing integration of AI tools directly into IDEs, exemplified by Cline in Visual Studio Code [22], and their application across the entire SDLC, points towards a future where AI functions as an intelligent assistant or "copilot" for both developers and testers. This collaboration has the potential to significantly accelerate development and testing cycles, leading to faster delivery of software [20]. For.NET MAUI development, this could translate into the rapid generation of XAML UIs and associated C# logic. However, the practical effectiveness of these AI assistants is profoundly dependent on several factors: the quality and breadth of their training data, the sophistication of prompt engineering techniques employed by the user, and the AI's nuanced understanding of complex contexts. This is particularly true for specialized frameworks like.NET MAUI, which possesses distinct XAML dialects and control patterns that differ from other XAML-based frameworks such as WPF or WinUI [31]. If an AI model, trained on a general corpus of code, fails to differentiate these subtleties, it may generate code that is syntactically plausible but functionally incorrect or non-performant within the MAUI environment, thereby negating the anticipated productivity gains. Consequently, the successful application of AI in.NET MAUI development, especially

for critical applications like FinTech test harnesses, hinges on the AI's fine-tuning or the developer's skill in crafting highly specific, context-rich prompts that guide the AI towards generating accurate and framework-compliant code.

A review of existing literature reveals a growing body of research into AI-assisted code generation [23], and the broader application of AI in software testing methodologies [34]. Concurrently, studies specific to.NET MAUI address its performance characteristics, inherent testing challenges, and recommended best practices [5]. The validation of FinTech SDKs is also a subject of focused research, emphasizing the necessity for comprehensive testing strategies that cover security, functionality, and regulatory compliance [12]. While the intersection of AI and.NET MAUI development is an emerging field, with some tools like Syncfusion's AI AssistView appearing [53], and discussions around using AI to aid in hiring MAUI developers, there is a discernible gap in specific research. This gap pertains to the documented use of contemporary generative AI models, accessed via integrated tools like Cline, for the explicit purpose of constructing.NET MAUI *test applications* aimed at *FinTech SDK validation*. Existing case studies that touch upon AI-generated XAML or C# often highlight the challenges AI faces with framework-specific nuances, indicating that a detailed investigation in this area is warranted [28]. This research seeks to address this gap by providing a detailed account of such an endeavor.

The primary purpose of this research is to systematically investigate and document the process of developing a.NET MAUI test application, targeting both Android and iOS platforms, through the use of AI-driven code generation. This involves leveraging a suite of AI models—Anthropic Claude, OpenAI's GPT series, and Google Gemini—accessed via the Cline extension in Visual Studio Code. The specific objective of this AI-generated test application is the validation of FinTech SDK functionalities, with a concentrated focus on simulating and testing wallet provisioning workflows.

The scope of this study encompasses several key phases. Initially, a.NET MAUI development environment will be established and integrated with the Cline extension, configured to access the selected AI models. Subsequently, the AI models will be prompted to generate the XAML UI and C# backend logic for the test application. This application will then be integrated with a representative FinTech SDK (which may be a mock or simulated version for research purposes), focusing on its wallet provisioning API calls. Following integration, automated tests, including both unit tests for business logic and UI tests (employing Appium with NUnit), will be developed and executed to validate the defined wallet provisioning workflow on Android emulators/devices and iOS simulators/devices. Throughout this process, the quality of the AI-generated code will be evaluated, and the overall efficacy of the AI-assisted development approach will be assessed. The study will also meticulously document any challenges encountered, effective prompt engineering strategies devised, and platform-specific considerations, such as the configuration of iOS entitlements.

The novelty of this research is centered on the specific, documented application of a combination of leading generative AI models (Claude, OpenAI, Gemini) facilitated by an integrated development environment tool (Cline), to construct a functional.NET MAUI test application. This application is explicitly designed for the complex task of validating FinTech SDKs. While AI for code generation and.NET MAUI development are independently known areas, their synergistic application for this particular, high-stakes testing purpose, accompanied by a detailed procedural journal of the experience, represents a unique contribution to the field. The selection of wallet provisioning as the target FinTech use case provides a concrete and relevant scenario, reflecting real-world challenges in mobile financial application testing.

Experimental Section/Material and Methods

The experimental procedure was designed to systematically evaluate the feasibility and effectiveness of using AI to generate a.NET MAUI test application for FinTech SDK validation. This section details the setup of the development environment, the AI models and tools used, the process of AI-assisted application generation, FinTech SDK integration, and the methodologies for test automation and validation.

- **Development Environment Setup**

The foundation of this research involved establishing a robust and correctly configured development environment capable of supporting.NET MAUI development and AI tool integration.

- **Visual Studio Configuration:**

Visual Studio Code (Version 1.93 or later) was selected as the primary Integrated Development Environment (IDE) due to its extensive support for.NET MAUI development through extensions and its compatibility with the Cline AI coding assistant. The ".NET Multi-platform App UI development" workload, typically associated with Visual Studio 2022, provides the necessary compilers, tools, and SDKs; these components were ensured to be available and correctly configured for command-line builds and VS Code integration.58 The Cline VS Code extension (latest available version during the study period) was installed and configured within VS Code.22 API keys for Anthropic Claude, OpenAI (GPT-4 and GPT-3.5-turbo), and Google Gemini models were obtained and configured within Cline. OpenRouter was considered and utilized where appropriate to streamline access to multiple models through a unified interface, simplifying

API key management and potentially offering access to a wider range of model versions.25

- **NET MAUI Environment:**

The.NET SDK (Version 8.0, aligning with contemporary.NET MAUI capabilities as suggested by sources like 2) was installed. The.NET MAUI workload was subsequently installed using the command dotnet workload install maui.

For Android development, Microsoft OpenJDK (Version 17 or compatible) and the Android SDK (latest stable version) were installed. Android emulators targeting various API levels (e.g., API 33, API 34) were configured using the Android Virtual Device (AVD) Manager. The Android environment within VS Code was verified using the .NET MAUI: Configure Android command and the Refresh Android environment option to ensure all paths and dependencies were correctly resolved [58]. For iOS development, a macOS machine was used as a build host. The latest stable version of Xcode compatible with the.NET MAUI workload was installed, along with the Xcode command-line tools and a selection of iOS simulators (e.g., iPhone 14, iPhone 15 running recent iOS versions).58 The iOS environment was verified in VS Code on the Mac using the .NET MAUI: Configure Apple command and Refresh Apple environment option.

The setup of a cross-platform mobile development environment, especially one that also integrates multiple AI services and their respective SDKs or API access mechanisms, is an inherently complex undertaking. It involves managing dependencies for.NET, MAUI, Java, Android SDK tools, Xcode, and the AI tools themselves. Each component has its own versioning and configuration requirements [25]. A failure at any point in this intricate setup chain—such as an incorrect SDK path, an incompatible version of a tool, or an issue with API key authentication for the AI models—can halt the entire development and testing process before the AI's code generation capabilities can even be utilized. This initial complexity represents a potential barrier to the widespread adoption of AI-assisted development, particularly for smaller teams or individual developers who may not have specialized environment configuration expertise. While AI tools aim to democratize and simplify the coding process itself, the "scaffolding" required to make these tools operational in a specific development context like.NET MAUI still demands significant technical proficiency and attention to detail. Future advancements in AI tooling might address this by offering more automated setup and configuration assistance for the entire development stack.

- AI Model Selection:

The selection of AI models was guided by their reported capabilities in code generation, understanding complex contexts, and their availability through the Cline extension.

- **Anthropic Claude (Opus and Sonnet versions):**

Chosen for their advanced reasoning capabilities, proficiency in handling complex coding tasks, and understanding of large codebases [15]. The Claude Code tool, often integrated or conceptually similar to Cline's interaction with Claude models, emphasizes deep codebase awareness and direct environment interaction [16].

- **OpenAI GPT (GPT-4 and GPT-3.5-turbo):**

Selected due to their widespread adoption, strong performance in general code generation, and extensive documentation regarding API interaction, which is relevant for generating C# logic for SDK communication.[17] Examples of using OpenAI for.NET MAUI applications, such as chat-like interfaces, provided a baseline understanding of their potential [17].

- **Google Gemini (Gemini 2.5 Pro and Flash versions):**

Included for their emerging capabilities in code generation, analysis of large codebases, and multimodal input potential, offering a comparative point to Claude and GPT models [19]. Their documented ability to generate code for various tasks, such as Python functions and Docker scripts, indicated potential for C# and XAML generation [68]. This multi-model approach allows for a comparative analysis of their strengths and weaknesses in the context of.NET MAUI development.

Table 1: AI Models and Cline Configuration Details

AI Model	Cline Version	Key Configuration Parameters (Example)	API Provider (via Cline)
Anthropic Claude 3 Opus	Latest	Temperature: 0.5, Max Tokens: 4096 (model default), Plan/Act Modes	Anthropic / OpenRouter
Anthropic Claude 3.7 Sonnet	Latest	Temperature: 0.5, Max Tokens: 8192 (model default), Plan/Act Modes	Anthropic / OpenRouter
OpenAI GPT-4	Latest	Temperature: 0.7, Max Tokens: 4096 (model default), Plan/Act Modes	OpenAI / OpenRouter
OpenAI GPT-3.5-turbo	Latest	Temperature: 0.7, Max Tokens: 4096 (model default), Plan/Act Modes	OpenAI / OpenRouter

AI Model	Cline Version	Key Configuration Parameters (Example)	API Provider (via Cline)
Google Gemini 2.0 Flash	Latest	Temperature: 0.6, Max Tokens: 8192 (model default), Plan/Act Modes	Google / OpenRouter
Google Gemini 2.5 Pro (Preview)	Latest	Temperature: 0.6, Max Tokens: 65536 (model default), Plan/Act Modes	Google / OpenRouter

*Note: "Latest" refers to the version of Cline available and models supported via Cline/OpenRouter during the study period (Q1-Q2 2025). Specific token limits and temperature settings were adjusted based on task complexity and observed output quality during iterative prompting. *

- **AI-Assisted Test Application Generation**

The core of the experimental work involved using the configured AI models via Cline to generate the.NET MAUI test application.

○ Methodology using Cline:

The development process heavily relied on Cline's distinct operational modes: "Plan" and "Act".22 The "Plan" mode was utilized for initial high-level design discussions with the AI, outlining the test application's architecture, the required pages for the wallet provisioning workflow (e.g., a page for card details input, a page for OTP entry, and a page to display the provisioning status), and the general UI flow between these pages. This collaborative planning phase allowed for refinement of the AI's understanding before code generation commenced. Subsequently, the "Act" mode was employed to instruct the AI to execute the agreed-upon plan, generating the.NET MAUI XAML markup for the UIs and the C# code for the corresponding backend logic and event handling. An iterative prompting approach was adopted. Initial prompts were high-level, requesting the basic structure of the application and its pages. These were then followed by more granular and detailed prompts for specific UI elements (e.g., Entry fields for card number, expiry date, CVV; Button controls for actions like "Submit" or "Verify OTP"; Labels for instructions and status messages) and the C# methods required to handle user interactions and orchestrate the workflow.

Context management was a critical aspect of this process. Cline's inherent capability to analyze the project's file structure and maintain context across interactions was leveraged. For more complex interactions or when switching between generating XAML and C#, strategies suggested by user experiences with Claude and Cline were employed, such as creating a cline_docs folder with relevant architectural notes or MAUI-specific patterns, and managing the length of interaction sessions to optimize token usage and maintain AI focus.

○ Prompt Engineering Strategies:

The success of AI-driven code generation is heavily reliant on the quality and precision of the prompts provided. Several established prompt engineering techniques were systematically applied:

■ **Clarity and Specificity:**

All prompts were formulated to be as clear and unambiguous as possible, providing explicit details regarding the target framework (.NET MAUI), programming languages (C# for logic, XAML for UI), target platforms (Android and iOS), and the desired structure and behavior of the generated components.[61] For instance, a prompt for a UI element would specify the control type, essential properties (like Placeholder text for an Entry, or Text for a Button), and layout containers (e.g., "Generate a.NET MAUI ContentPage in XAML for wallet provisioning. The page should be titled 'Enter Card Details'. Include an Entry for card number with AutomationId='CardNumberEntry', an Entry for expiry date (MM/YY) with AutomationId='ExpiryDateEntry', an Entry for CVV with AutomationId='CvvEntry', and a Button labeled 'Provision Wallet' with `AutomationId='ProvisionButton". Use a VerticalStackLayout for arrangement with appropriate spacing.").

■ **Persona Adoption:**

Prompts often included instructions for the AI to adopt the persona of an "expert.NET MAUI developer" or a "specialist in cross-platform mobile UI design" to encourage the generation of code adhering to best practices and framework conventions [61].

■ **Few-Shot Prompting:**

When initial AI generations were suboptimal or did not conform to desired.NET MAUI patterns (e.g., incorrect XAML syntax, inappropriate control usage), few-shot prompting was employed. This involved providing the AI with small, correct examples of the desired XAML structure or C# code snippets to guide subsequent generations [72].

■ **Chain-of-Thought (CoT) Prompting:**

For more complex components, such as a multi-step UI flow or intricate C# logic for state management, CoT prompting was used. The task was broken down into smaller, sequential steps, and the AI was asked to "think step by step" or to generate code for each step iteratively [72]. An example for the wallet provisioning logic: "First, design the XAML for the card input form page. Second, generate the C# code-behind for this page to handle input validation for the card number, ensuring it is numeric and of a typical length. Third, create a C# method within the code-behind that, upon button click,

will call the (mocked) FinTech SDK's provision API with the collected card details."

■ **Reference Text/Context Provision:**

To aid the AI, snippets of mock FinTech SDK documentation (detailing API endpoints and request/response payloads) or examples of preferred.NET MAUI coding patterns (e.g., using async/await for service calls, basic data binding, or specific MAUI Essentials features) were included in prompts where relevant.[71] While a full MVVM pattern was considered too complex for the initial AI generation of a simple test app, prompts encouraged separation of concerns where feasible.

■ **Iterative Refinement:**

AI-generated code was consistently reviewed. Feedback was provided to the AI for refinement. This was particularly crucial given the known challenges AI models face with the specific nuances of.NET MAUI's XAML dialect and control set, which differ from WPF or WinUI. Prompts were adjusted based on the AI's output, for example: "The previously generated XAML for the ListView does not use compiled bindings; please regenerate using x:DataType and compiled binding syntax for better performance," or "The C# method for the API call does not include comprehensive error handling for network issues; please add try-catch blocks and log exceptions." [71].

■ **Specifying Output Format:**

When specific output structures were needed, particularly for code snippets or configuration files, prompts included instructions like "Provide the C# code within a csharp... block" or used XML tags to delineate expected output sections, as suggested for Claude 4 [61].

○ Baseline App Structure Understanding:

The AI models were prompted to generate an application structure consistent with standard.NET MAUI project organization. This includes a main App.xaml for global resources, an AppShell.xaml for defining the basic navigation structure (e.g., using TabBar for different stages of the wallet provisioning workflow), and multiple ContentPage XAML files for each distinct screen (e.g., card input, OTP verification, success/failure display). The C# code-behind for these pages was expected to handle UI event logic and interactions with service classes. This structural approach is similar to that demonstrated in official.NET MAUI tutorials, such as the "Notes" application example 66, which provides a good reference for typical MAUI app construction involving XAML for UI elements (Entry, Button, Label, ListView/CollectionView) and C# for event handlers and business logic. The quality of AI-generated.NET MAUI code demonstrates a direct correlation with the specificity and contextual richness embedded in the prompts. This is particularly evident given.NET MAUI's unique XAML features, control sets (e.g., Picker instead of ComboBox, VerticalStackLayout instead of StackPanel found in WPF), and distinct styling mechanisms, including its support for CSS-like styling. AI models, primarily trained on vast and diverse codebases, may not possess an inherently deep or nuanced understanding of every framework's specific intricacies unless explicitly guided or fine-tuned on framework-specific data.[31] Consequently, generic prompts such as "create a page with a dropdown list" are likely to yield XAML or C# code that is either non-functional within a MAUI context, uses incorrect control types, or implements outdated patterns. This necessitates significant manual rework, thereby diminishing the anticipated productivity benefits of AI assistance. Effective prompt engineering, potentially incorporating few-shot learning with correct.NET MAUI XAML and C# examples, becomes a critical factor for success. This research meticulously documented the iterative process of prompt design and refinement to achieve usable code.

Table 2: Prompt Engineering Examples for Key Test App Components

Component	Initial Prompt Example (Simplified)	Refined Prompt Example (Simplified, after iteration)	AI Model Used (Example)	Brief Notes on Outcome
CardInputPage.xaml	"Create a MAUI XAML page for card input."	"Generate a.NET MAUI ContentPage XAML for 'CardInputView' with VerticalStackLayout. Include: Label 'Card Number', Entry (AutomationId='CardNumEntry', Keyboard='Numeric'), Label 'Expiry (MM/YY)', Entry (AutomationId='ExpiryEntry', Placeholder='MM/YY'), Label 'CVV', Entry (AutomationId='CvvEntry', Keyboard='Numeric', IsPassword='True'), Button (AutomationId='SubmitCardButton', Text='Submit'). Ensure proper spacing."	Claude 3 Sonnet	Initial prompt was too vague. Refined prompt yielded a usable basic structure, but styling and detailed validation attributes needed further prompts.

Component	Initial Prompt Example (Simplified)	Refined Prompt Example (Simplified, after iteration)	AI Model Used (Example)	Brief Notes on Outcome
SDK Service Call C#	"Write C# to call a provision API."	"Generate a public async Task<WalletProvisionResponse> ProvisionWalletAsync(string cardNumber, string expiry, string cvv) method in C#. Use HttpClient to POST JSON to 'https://api.mockfintech.com/provision'. Include try-catch for HttpRequestException. Define WalletProvisionResponse and request classes. Use.NET MAUI best practices for async calls."	OpenAI GPT-4	Initial prompt lacked specifics. Refined prompt produced a more complete method stub, including basic error handling and model definitions. Needed further refinement for header injection and detailed error parsing.
OTPPage.xaml	"MAUI XAML for OTP entry."	"Create a.NET MAUI ContentPage XAML 'OtpView'. Display a Label 'Enter OTP received via SMS'. Add an Entry (AutomationId='OtpEntry', Keyboard='Numeric', MaxLength='6'). Add a Button (AutomationId='VerifyOtpButton', Text='Verify OTP'). Center elements using VerticalStackLayout."	Google Gemini Pro	Gemini produced a functional layout. Prompting for Max Length and Keyboard type was effective.

- **FinTech SDK Integration**

A crucial part of the test application is its ability to interact with a FinTech SDK, specifically for the wallet provisioning workflow.

○ Selection of FinTech SDK:

For this research, a mock FinTech SDK was defined to ensure reproducibility and to avoid dependencies on proprietary or access-restricted commercial SDKs. This mock SDK exposes a set of RESTful API endpoints representative of a typical wallet provisioning service. The defined API endpoints are:

1. POST /wallet/provision
   - Request Body: {"cardNumber": "string", "expiryDate": "string (MM/YY)", "cvv": "string", "userName": "string"}
   - Response Body (Success): {"provisioningId": "string", "status": "PENDING_OTP"}
   - Response Body (Error): {"errorCode": "string", "errorMessage": "string"}

2. POST /wallet/confirm Provisioning
   - Request Body: {"provisioningId": "string", "otp": "string"}
   - Response Body (Success): {"walletId": "string", "status": "PROVISIONED", "card Masked Number": "string"}
   - Response Body (Error): {"errorCode": "string", "errorMessage": "string"}

3. GET /wallet/{walletId}/status
   - Response Body (Success): {"status": "string", "cardMaskedNumber": "string"}
   - Response Body (Error): {"errorCode": "string", "errorMessage": "string"} A simple local mock server (e.g., using Node.js with Express, or a.NET Minimal API) was set up to respond to these endpoints with predefined success and error scenarios.

○ Integration Process:

The AI models were prompted to generate C# service classes and methods to encapsulate HTTP calls to these mock FinTech SDK endpoints. Prompts emphasized the use of Http Client, asynchronous programming patterns (async/await), strongly-typed request and response models (POCOs), and robust error handling mechanisms (e.g., try-catch blocks for Http Request Exception, checking HTTP status codes). Example prompt: "Generate a C# service class named 'Wallet Api Service' with a method public async Task<Wallet Provision Response> Provision Card Async (Wallet Provision Request request Data). This method should use Http Client to send a POST request to the '/wallet/provision' endpoint of a base URL (configurable). Serialize request Data to JSON for the request body. Deserialize the JSON response into a Wallet Provision Response object. Implement error handling for network issues and non-success HTTP status codes, throwing a custom Api Exception with details." The generated service methods were then integrated into the AI-generated.NET MAUI application's C# code-behind files or, where appropriate, into simple View Model classes associated with the respective pages.

○ Platform-Specific Configurations for FinTech SDKs:

While the mock SDK primarily relies on standard HTTPS communication, considerations for platform-specific configurations relevant to real-world FinTech SDKs were explored:

- **iOS Entitlements:** For a production FinTech app, secure storage of sensitive data like API keys or session tokens is critical. Prompts were formulated to ask the AI about necessary iOS entitlements for such scenarios, for example, "If a.NET MAUI iOS app needs to securely store an API token received from a FinTech SDK, what entitlements should be configured in Entitlements. plist and how would one typically use the Keychain service?" This led to discussions around keychain-access-groups if sharing credentials between apps by the same vendor was a consideration, or general secure storage practices. For the mock SDK, direct use of complex entitlements like Apple Pay was not in scope, but understanding AI's awareness of these was part of the exploration. App Attest was also considered as a relevant entitlement for enhancing security in real FinTech scenarios.
- **Android Permissions:** Similarly, for Android, the AI was prompted regarding necessary permissions in AndroidManifest.xml. For basic API calls, the INTERNET permission is fundamental. Example prompt: "What Android permissions are required in AndroidManifest.xml for a.NET MAUI app that makes HTTPS requests to a backend API and might need to access network state?"

The integration of any FinTech SDK, even a simulated one, invariably introduces dependencies on secure data handling protocols and often necessitates leveraging platform-specific security features, such as the iOS Keychain or Android Keystore for persistent, secure storage of tokens or sensitive configuration data. AI-generated code intended for these SDK interactions must be meticulously scrutinized to ensure adherence to security best practices. AI models, while capable of generating functional code [15], might not inherently prioritize these security considerations or implement them correctly without explicit, detailed, and context-aware prompting. For instance, an AI might generate code that inadvertently logs sensitive data, stores tokens insecurely (e.g., in plain text preferences), or fails to implement proper input validation for data being sent to the SDK, unless specifically instructed otherwise. A failure to guide the AI with security-centric prompts could result in an AI-generated test application that, paradoxically, introduces security vulnerabilities, thereby undermining its primary purpose of validating a FinTech SDK in a secure and representative manner. This highlights the indispensable role of human expert review and security-focused testing, even for AI-generated test tools.

- **Test Automation and Validation Methodology**

A comprehensive validation strategy was employed, combining unit tests for backend logic and UI automation tests for end-to-end workflow verification.

○ Wallet Provisioning Workflow Definition:

The core workflow targeted for validation was defined as follows:

1. **Card Input:** The user navigates to the CardInputPage, enters card details (card number, expiry date, CVV), and submits the form.
2. **Initial Provisioning Call:** The application calls the mock FinTech SDK's /wallet/provision API endpoint with the provided card details.
3. **OTP Navigation:** If the API response indicates successful initiation (e.g., status: "PENDING_OTP"), the application navigates to the OTP Page, displaying any relevant information (like provisioning Id for internal tracking).
4. **OTP Entry:** The user enters the (mock) OTP received on the OTP Page and submits.
5. **Provisioning Confirmation Call:** The application calls the mock SDK's /wallet/confirm Provisioning API endpoint with the provisioning Id and the entered OTP.
6. **Status Display:** The application navigates to a Status Page, which displays the final provisioning status (success or failure) based on the API response, potentially showing the masked card number upon success.

○ Unit Testing:

Unit tests were developed for the C# service classes responsible for interacting with the mock FinTech SDK. xUnit was chosen as the testing framework due to its widespread use in the.NET ecosystem and compatibility with.NET MAUI projects. These tests focused on:

- Verifying correct construction of HTTP requests (method, URL, headers, JSON body).
- Validating proper parsing of JSON responses into C# model objects.
- Ensuring robust error handling for various scenarios, such as network errors, API-returned error codes (e.g., invalid card, insufficient funds), and unexpected response formats. The AI tools (specifically, direct prompting of models like GPT-4 or Claude, and exploring tools like CodiumAI if integrated via Cline) were tasked with generating initial unit tests for these service methods. Example prompt: "Generate xUnit tests for the C# ProvisionCardAsync method in the WalletApiService class. Include test cases for a successful 200 OK response, a 400 Bad Request

API error, a 500 Internal Server Error, and a network connectivity issue (HttpRequestException). Mock HttpClient using MOQ or a similar library."

- UI Automation Testing:

Appium (Version 2.0+) was selected for cross-platform UI test automation, in conjunction with the N Unit testing framework for test definition and execution.5

- **UI Element Identification:** A key strategy for reliable UI automation is the consistent use of AutomationId properties for UI elements in XAML. The AI models were explicitly prompted to include unique Automation Ids for all interactable elements (e.g., Entry, Button, Labels displaying dynamic data) in the generated XAML files [79]. Example prompt fragment: "...ensure the CVV Entry has AutomationId='CvvEntry' and the Submit button has AutomationId='SubmitButton'."
- **Test Scenarios:** UI automation scripts were developed to cover the end-to-end wallet provisioning workflow on both Android emulators (e.g., Pixel 5, API 33) and iOS simulators (e.g., iPhone 14, latest iOS). Scenarios included:
- Successful wallet provisioning with valid card details and OTP.
- Failure due to invalid card number format (client-side validation if implemented, or server-side error).
- Failure due to incorrect CVV.
- Failure due to incorrect OTP.
- Handling of API errors during the provisioning or confirmation steps. The Appium test scripts were written in C# using the Appium.NET driver.

- Device Provisioning for iOS Testing:

While simulators are generally adequate for API-based testing, if the FinTech SDK had interactions with device-specific hardware (e.g., Secure Enclave for key storage, NFC chip), testing on a physical iOS device would be necessary. This would involve the manual provisioning process: registering the device UDID in the Apple Developer Account, creating an App ID (explicit or wildcard), generating development certificates, and creating a development provisioning profile that includes the App ID, certificates, and registered devices. This profile would then be downloaded and configured in Visual Studio for deploying the app to the physical device.80 for this study, given the mock SDK's nature, simulator testing was deemed sufficient, but the process was noted.

- Metrics for Evaluating AI-Generated Code Quality:

A multi-faceted approach was used to evaluate the quality of the.NET MAUI code generated by the AI models:

- **Functional Correctness:** Primarily assessed through the execution of the automated unit and UI tests. Success or failure of these tests provided direct evidence of whether the AI-generated UI and logic performed the intended operations correctly for the defined wallet provisioning workflow [81], manual exploratory testing was also conducted to catch issues not covered by automated tests.
- **Code Quality Metrics (Static Analysis):** The generated C# and XAML code was subjected to static analysis using tools integrated into Visual Studio (e.g.,.NET Analyzers, Roslyn Analyzers) and potentially external linters if applicable. Metrics tracked included:
- Adherence to C# coding standards and XAML best practices.
- Cyclomatic complexity of key generated C# methods (e.g., event handlers, service call methods) to gauge maintainability [81].
- Identification of "code smells" such as duplicated code blocks, overly long methods, or deeply nested conditional logic [81].
- **Security Vulnerabilities (Conceptual Analysis):** While a full dynamic security scan was beyond the scope for AI-generated client code, a conceptual analysis was performed. Generated C# snippets, particularly those handling user input or interacting with the mock SDK, were reviewed against common vulnerabilities (e.g., lack of input sanitization if data were to be displayed elsewhere, insecure storage of hypothetical sensitive data if prompted broadly). The CodeVulnerabilityEvaluator from Microsoft. Extensions. AI.Evaluation libraries was considered as a potential tool for future, more in-depth analysis of AI-generated server-side or more complex client-side logic, though its direct application to this specific MAUI client app's generated code was primarily a conceptual exercise [82].
- **Maintainability & Readability:** A subjective assessment was conducted by the primary researcher (acting as an experienced mobile test automation engineer) regarding the ease of understanding, modifying, and debugging the AI-generated code. Factors considered included code structure, clarity of variable and method names, presence and quality of comments (if generated), and overall logical flow.
- **Efficiency of Generation (Qualitative):** The perceived time taken for the AI to generate code for specific components was noted and qualitatively compared against an estimate of manual effort for a developer familiar

with.NET MAUI.

- **Amount of Manual Rework:** A critical metric was the percentage of AI-generated lines of code (LoC) that required modification. Edits were categorized as:
- None (usable as-is).
- Minor (e.g., syntax correction, renaming, minor logic adjustment).
- Major (e.g., significant refactoring, complete rewrite of a method or XAML section).

RESULTS AND DISCUSSION

This section presents the findings from the AI-assisted development of the.NET MAUI test application and the subsequent validation of the mock FinTech SDK's wallet provisioning workflow.

- **Summary of AI-Assisted Test Application Development Process**

The development of the.NET MAUI test application was undertaken primarily using the Cline extension in Visual Studio Code, orchestrating interactions with Anthropic Claude (Sonnet and Opus), OpenAI GPT-4, and Google Gemini Pro models. The process began with high-level planning prompts in Cline's "Plan" mode to define the application structure, pages (CardInputPage, OtpPage, StatusPage), and basic navigation flow for the wallet provisioning workflow. This collaborative planning phase proved useful for refining the AI's understanding of the requirements before code generation. Subsequently, Cline's "Act" mode was used to generate XAML for UI layouts and C# for code-behind logic and service interactions. Iterative prompting was essential. For instance, initial XAML generations often required refinement for MAUI-specific syntax, layout manager choices (e.g., preferring VerticalStackLayout or Grid over potentially less performant or less MAUI-idiomatic structures), and the inclusion of AutomationId attributes for UI testing.

Comparing the AI models:

- **Anthropic Claude 3 Sonnet/Opus:** Generally excelled at understanding more complex prompts and generating coherent blocks of C# code, especially for service logic involving async/await and basic data models. Its ability to maintain context over longer interactions was noticeable, though still requiring careful management [15]. Opus, when accessible, provided more nuanced solutions for complex logic prompts.
- **OpenAI GPT-4:** Proved highly effective for generating both XAML and C# snippets. It was particularly adept at scaffolding UI pages based on descriptions of elements and layout. Its responses were often syntactically correct for.NET MAUI, but sometimes required explicit reminders to use MAUI-specific controls or attributes if the prompt was not sufficiently precise [17].
- **Google Gemini Pro:** Showed good capability in generating C# helper functions and basic XAML structures. It responded well to structured prompts detailing specific attributes for UI elements [19]. However, for more complex MAUI page layouts or intricate C# logic, it sometimes required more iterations or more explicit examples (few-shot prompting) compared to GPT-4 or Claude.

A general observation was the challenge AI models faced in consistently distinguishing.NET MAUI XAML from other XAML dialects like WPF or older Xamarin.Forms syntax, especially if prompts were not highly specific. For example, requests for a "ComboBox" might yield WPF/WinUI XAML, requiring re-prompting for a MAUI Picker.Qualitatively, the AI-assisted approach significantly reduced the initial time for creating basic page structures and C# method stubs. However, the time spent on refining prompts, reviewing generated code, and making MAUI-specific corrections constituted a substantial portion of the development effort. This "AI-assisted debugging" phase was particularly notable for XAML, where layout nuances and control-specific properties often required manual adjustments.

- **Evaluation of AI-Generated.NET MAUI Code Quality**

The quality of the AI-generated code was assessed using the metrics defined in the methodology.

- **Functional Correctness:** The AI-generated UI elements, after necessary refinements, rendered correctly on both Android and iOS. Basic C# logic for button clicks and placeholder navigation (before full SDK integration) generally worked as prompted, provided the prompts were sufficiently detailed. The core functionality of the test app for the wallet provisioning workflow was achieved after integrating and debugging the AI-generated components.

- **Code Quality Metrics (Static Analysis):**
  - NET Analyzers in Visual Studio flagged some issues in initial generations, mostly related to unused variables, potential null reference exceptions (if error handling was not explicitly prompted for), and minor styling suggestions.
  - Cyclomatic complexity for AI-generated C# event handlers and service methods was generally low to moderate for straightforward tasks. However, for more complex logic requested in a single prompt, the AI sometimes produced longer methods that benefited from manual refactoring into smaller, more focused functions.
  - "Code smells" like minor code duplication were occasionally observed if similar UI interaction

patterns were requested for different pages without explicitly prompting for reusable helper methods.

- Regarding.NET MAUI best practices, AI models did not consistently apply optimizations like compiled bindings (x:DataType) unless specifically prompted. Layout generation sometimes resulted in deeper nesting than necessary, which could impact performance on resource-constrained devices, requiring manual simplification.
- **Security (Conceptual Analysis):** For C# snippets related to (mock) SDK interaction, initial broad prompts (e.g., "generate code to call an API") did not inherently include robust security practices like secure credential handling. Prompts had to be very specific to guide the AI towards even conceptual secure coding patterns. A Code Vulnerability Evaluator [82] would likely flag generic API call code if it handled sensitive data without explicit security measures. This underscores the need for expert review of AI-generated code in security-sensitive contexts like FinTech.
- **Maintainability & Readability:** The readability of AI-generated code varied. Simpler C# methods and XAML blocks were generally clear. More complex generations sometimes lacked sufficient comments (unless prompted) or used generic variable names that required manual improvement for better maintainability.
- **Manual Rework:** Approximately 40-60% of the AI-generated XAML required minor to major edits, primarily for MAUI-specific attribute corrections, layout adjustments for responsiveness, and styling. For C# logic, about 30-50% required rework, focusing on error handling, integration points, and adherence to project-specific conventions or more robust patterns. Pure boilerplate (e.g., class definitions, basic method signatures) often required minimal changes.

The evaluation suggests that while AI can generate functionally "correct" code that passes initial tests, this code may not always adhere to the latest framework best practices or exhibit high maintainability without careful prompting and expert review. AI models learn from vast datasets of existing code. If this training data includes common but suboptimal patterns (e.g., overly nested layouts for visual design simplicity, or older C# asynchronous patterns), the AI may replicate these. For a rapidly evolving framework like.NET MAUI, whose best practices are continuously refined, AI-generated code might reflect an amalgamation of patterns from various stages of the framework's evolution or from related but distinct XAML frameworks. This means that "functional correctness" as determined by basic tests is an insufficient sole metric for AI-generated code quality; adherence to current best practices, performance considerations, and long-term maintainability are equally, if not more, critical, especially for enterprise-grade applications.

Table 3: AI-Generated Code Quality Evaluation Summary (Illustrative Examples)

AI Model	Generated Component	Functional Correctness	Static Analysis Issues (Count)	Cyclomatic Complexity (Avg)	Security Concerns (Conceptual)	Manual Rework (%)	Readability Score (1-5, 5=High)
Claude 3 Sonnet	CardInputPage.xaml	Partial (Layout issues)	3 (minor styling)	N/A	0	60%	3
OpenAI GPT-4	CardInputPage.xaml.cs (event handlers)	Pass (basic logic)	1 (unused import)	3	0	40%	4
Gemini Pro	WalletApiService.cs (method stubs)	Pass (signatures only)	0	1	1 (prompted for error handling, initially missed some cases)	50%	3
Claude 3 Opus	WalletApiService.cs (full methods with error handling)	Pass	0	5	0 (after specific security prompts)	30%	4

- **Detailed Results of FinTech SDK (Wallet Provisioning) Validation**

The AI-generated.NET MAUI application, after necessary refinements, was used to execute tests against the mock FinTech SDK.

- **Unit Tests:** Unit tests for the WalletApiService class (interacting with the mock SDK) were largely successful after initial AI generation and subsequent manual refinement. AI (GPT-4) was able to generate basic xUnit test structures and some test cases for success paths and simple error conditions. More complex scenarios, like mocking HttpClient responses for specific error codes or testing nuanced exception handling, required significant manual input or highly detailed prompts. Overall, AI assistance reduced the initial setup time for unit tests by an estimated 20-30%.

- **UI Automation Tests (Appium):**
  - **Android:** UI automation tests for the wallet provisioning workflow executed successfully on Android emulators (Pixel 5, API 33 and API 34). Appium was able to reliably identify UI elements using the AI-prompted AutomationIds. The workflow involving card input, OTP submission, and status verification passed for success scenarios and correctly identified failures for invalid data/OTP scenarios.
  - **iOS:** Tests on iOS simulators (iPhone 14, iOS 16.x and iPhone 15, iOS 17.x) also generally passed. Some minor timing adjustments in Appium scripts were occasionally needed due to differences in UI rendering speed or animation transitions compared to Android. No specific iOS entitlement issues were encountered for the mock SDK's functionality, as it relied on basic internet access, which is implicitly allowed. Had the SDK required Keychain access, Entitlements. plist configuration would have been critical.

- Platform-Specific Issues:

Minor UI rendering differences were observed between Android and iOS for some complex layouts initially generated by AI, requiring manual XAML adjustments to ensure visual consistency. These were typically related to default padding/margin of controls or layout container behavior. NET MAUI's abstraction layer handled most differences, but pixel-perfect consistency sometimes needed explicit styling.

- **Performance:** The mock SDK API calls from the MAUI app were performant, with response times primarily dictated by the mock server's latency. The UI remained responsive during these asynchronous operations due to the correct use of async/await in the AI-generated (and refined) C# code.

Table 4: Wallet Provisioning Workflow Test Execution Results (Android & iOS - Aggregated)

Test Case ID	Description	Expected Result	Actual Result (Android)	Actual Result (iOS)	Pass/Fail	Notes
WP_TC001	Successful Provisioning (Valid Card & OTP)	Wallet Provisioned, Masked Card Displayed	Wallet Provisioned, Masked Card Displayed	Wallet Provisioned, Masked Card Displayed	Pass	
WP_TC002	Invalid Card Number (Format)	Error Message: Invalid Card	Error Message: Invalid Card	Error Message: Invalid Card	Pass	Client-side validation (prompted) and server error tested.
WP_TC003	Provisioning API Error (Server Down)	Error Message: Service Unavailable	Error Message: Service Unavailable	Error Message: Service Unavailable	Pass	
WP_TC004	Invalid OTP	Error Message: Invalid OTP	Error Message: Invalid OTP	Error Message: Invalid OTP	Pass	
WP_TC005	OTP Timeout (Simulated)	Error Message: OTP Expired	Error Message: OTP Expired	Error Message: OTP Expired	Pass	Mock server simulated timeout.

- **Discussion of Challenges Encountered**

Several challenges emerged during this research, spanning AI code generation, SDK integration, and cross-platform testing.

- **AI Code Generation Challenges:**
  - **Framework Specificity:** A primary challenge was guiding the AI models to generate code that was not just syntactically correct C# or XAML, but specifically idiomatic and functional.NET MAUI code. As noted by other developers, AI models often confused.NET MAUI XAML with WPF, WinUI, or older Xamarin. Forms syntax, leading to errors in control usage (e.g., TextBlock vs. Label, StackPanel vs. StackLayout), property names, or event handling. This required highly specific prompts, often including explicit mentions of ".NET MAUI" and sometimes providing few-shot examples of correct MAUI syntax.
  - **Complex UI and State Management:** While AI was proficient at generating simpler UI layouts and event handlers, prompting for more complex UI interactions (e.g., dynamic visibility changes based on state, custom animations) or robust client-side state management for the multi-step wallet provisioning flow proved more difficult and often resulted in code requiring substantial manual refactoring.
  - **Contextual Understanding:** The AI models, despite their advancements, showed limitations

in deeply understanding the FinTech domain's implicit requirements (e.g., security nuances, typical user expectations for financial workflows) or the specific constraints of the mock SDK without extensive, detailed context provided in the prompts.

- **AI Hallucinations:** Occasionally, models would generate plausible-sounding but incorrect code, such as inventing non-existent.NET MAUI properties or methods, or misinterpreting API documentation for the mock SDK.
- **Token Limits and Cost:** For generating larger files or engaging in extended iterative refinement with more powerful models like GPT-4 or Claude Opus, managing API token limits and associated costs became a practical consideration. Breaking down generation tasks into smaller chunks was often necessary.

- **SDK Integration Challenges:**
  - **Generated Stubs vs. Real Logic:** Integrating the AI-generated C# service stubs with the application logic sometimes revealed discrepancies in assumed data contracts or error handling patterns, requiring manual alignment.
  - **Secure Data Handling:** Ensuring that even mock sensitive data (like card numbers) was handled appropriately within the test app (e.g., not logged excessively, cleared from memory after use) required careful review of AI-generated code, as AI didn't prioritize these aspects by default.
  - **Platform Configurations:** While basic INTERNET permissions were straightforward, prompting AI for correct iOS entitlement configurations (e.g., for hypothetical Keychain usage) was less reliable, often yielding generic advice rather than precise Entitlements. plist snippets.

- **Cross-Platform Testing with.NET MAUI Challenges:**
  - **UI Inconsistencies:** Despite.NET MAUI's goal of UI unification, subtle differences in rendering or behavior of certain controls or layouts between Android and iOS occasionally surfaced during UI automation, requiring platform-specific tweaks in Appium scripts or conditional XAML [14].
  - **Appium Stability and Setup:** Setting up Appium for.NET MAUI and ensuring stable script execution across both platforms involved typical mobile automation challenges, such as managing Appium server instances, driver versions, and locator strategies.
  - **Build and Deployment Times:** Slow build times, particularly for iOS, and the overhead of deploying to emulators/simulators, sometimes impacted the iteration speed of testing AI-generated UI changes.

The process of debugging and refining AI-generated code, especially for a framework with specific nuances like.NET MAUI, can introduce a new type of bottleneck. While AI can rapidly produce initial code drafts [20], if this code is misaligned with framework specifics (e.g., using incorrect XAML controls, misunderstanding MAUI's layout system, navigation, or platform handler architecture), the developer or tester must then invest considerable time in diagnosing and correcting these AI-induced errors. This "AI-assisted debugging" phase can be substantial, particularly for complex UIs or when the AI's training data lacks sufficient high-quality, up-to-date.NET MAUI examples. In some instances, the effort required to fix flawed AI-generated code might approach the effort of writing it manually, especially if the developer needs to first understand the AI's (potentially flawed) logic before correcting it. This highlights a critical trade-off: the speed of initial generation versus the time spent on subsequent validation and correction.

- **Interpretation of Findings**

The research indicates that Cline, when used with a combination of advanced AI models like Claude, GPT-4, and Gemini, serves as a capable orchestrator for AI-assisted.NET MAUI development within Visual Studio Code. It effectively streamlines the process of sending prompts and receiving code snippets directly in the IDE. In terms of comparative effectiveness for.NET MAUI code generation:

- **Anthropic Claude models (Opus and Sonnet)** demonstrated strong capabilities in generating more complex C# logic and maintaining context over longer, iterative prompting sessions. They were often better at grasping the overall intent of a multi-step task.
- **OpenAI GPT-4** showed excellent proficiency in generating both XAML layouts and C# code-behind, particularly for well-defined UI components and event handlers. Its XAML output was often closer to MAUI conventions with precise prompting.
- **Google Gemini Pro** was effective for generating smaller, focused C# methods and basic XAML structures, but typically required more explicit guidance or examples for complex MAUI-specific features.

The AI-generated test application, after the necessary manual refinements and debugging, successfully facilitated the validation of the mock FinTech SDK's wallet provisioning workflow on both Android and iOS. The automated unit and UI tests were able to execute the defined scenarios and verify the expected outcomes. The current maturity of AI tools for building non-trivial.NET MAUI applications can be described as "promising but requires significant expert

guidance." While AI can accelerate the generation of initial code structures and boilerplate, achieving production-quality, performant, and maintainable MAUI code still necessitates considerable human intervention, particularly for UI fine-tuning, framework-specific optimizations, and robust error handling. The AI acts more as a "junior developer on steroids" – capable of producing code quickly but needing senior oversight and correction. The performance of the mock FinTech SDK interactions within the.NET MAUI app was satisfactory, with UI responsiveness maintained through proper asynchronous programming patterns, which AI was generally capable of generating when prompted.

- **Speculations on Broader Applicability**

The AI-assisted approach demonstrated in this study for generating a test application for wallet provisioning could likely be extended to test other complex FinTech SDK functionalities. With appropriate prompting and context, AI models could assist in creating test harnesses for features such as Know Your Customer (KYC) document submission and verification flows, payment processing (initiating payments, handling different payment methods, processing refunds), or secure data exchange mechanisms. Furthermore, there is potential for AI to assist in generating more sophisticated test scenarios. For instance, AI could be prompted to devise edge-case scenarios for wallet provisioning based on common failure points in financial systems or to generate varied test data (e.g., different card types, regional formats). AI could also play a role in generating initial drafts of security-focused test cases, such as attempting to bypass OTP mechanisms or inject malformed data into SDK API calls, although such tests would require extensive security expertise for proper design and validation.

- **Limitations of the Current Study**

This research, while providing valuable insights, has several limitations:

  - **Mock SDK:** The use of a mock/simulated FinTech SDK, while necessary for control and reproducibility, does not capture the full spectrum of complexities, potential undocumented behaviors, or stringent security requirements of a production-grade commercial FinTech SDK. Real-world SDKs often have more intricate authentication mechanisms, state management, and error conditions.
  - **Limited Workflow Scope:** The study focused on a specific wallet provisioning workflow. A comprehensive FinTech application involves numerous other workflows and edge cases that were not explored.
  - **Subjectivity in Code Quality Evaluation:** Some aspects of AI-generated code quality, such as maintainability and readability, were assessed subjectively by the researcher. While static analysis metrics provide objectivity, the overall "goodness" of code involves human judgment.
  - **Specific AI Tools and Models:** The findings are based on the specific versions of Anthropic Claude, OpenAI GPT, Google Gemini, and the Cline extension available during the research period. The rapidly evolving nature of AI means that newer models or different tools might yield different results.
  - **Single Researcher Bias:** The prompt engineering and code evaluation were primarily conducted by a single researcher, which could introduce bias in terms of prompting style and assessment criteria.

- **Future Prospects and Potential Research Directions**

The findings of this study open several avenues for future research in the intersection of AI,.NET MAUI, and FinTech SDK testing:

  - **Advanced and Fine-Tuned AI Models:** Investigating the capabilities of future AI models (e.g., next-generation Claude, GPT, Gemini) that may possess an improved intrinsic understanding of.NET MAUI specifics or could be fine-tuned on curated.NET MAUI-specific datasets to enhance the quality and accuracy of generated code [43].
  - **AI for Platform Configuration:** Researching AI-driven techniques for automatically identifying and suggesting necessary iOS entitlements or Android permissions based on an analysis of FinTech SDK documentation or functionality descriptions.
  - **AI-Powered Test Data Generation:** Exploring the use of AI to generate more diverse, realistic, and contextually relevant test data for various FinTech scenarios, including edge cases and security testing inputs.
  - **Self-Healing Test Automation Scripts:** Investigating the application of AI to maintain and self-heal Appium test automation scripts for.NET MAUI applications, where AI could adapt scripts to UI changes or automatically update locators [34].
  - **AI in Security and Compliance Testing:** Further exploring the role of AI in generating security test cases and assisting in compliance checks for AI-generated test applications used in the FinTech domain, ensuring that the test tools themselves do not introduce vulnerabilities [10].

The overarching evolution of AI in software development and testing points towards an "AI-native SDLC," where AI is not merely an auxiliary tool but an integral component woven into each phase of the lifecycle. In the context of FinTech SDK validation, this trajectory suggests a future where AI's role could expand significantly. Imagine an AI system that not only

generates the.NET MAUI test application but also parses the FinTech SDK's technical documentation to identify key API endpoints relevant to a workflow like wallet provisioning. Based on this understanding, it could then suggest appropriate MAUI UI flows to effectively test these APIs, proceed to generate the MAUI application code and corresponding Appium test scripts, and even cross-reference the SDK's functionality against common security vulnerability patterns or regulatory compliance checklists pertinent to financial services (e.g., PCI DSS requirements for card data handling). This implies a profound shift in the role of the mobile test automation engineer, moving from detailed script authorship towards AI orchestration, prompt refinement, rigorous validation of AI-generated artifacts, and strategic test planning. Such a system could dramatically accelerate the testing of complex FinTech solutions while potentially enhancing test coverage and consistency.

CONCLUSION

This research systematically documented the process of leveraging generative AI—specifically Anthropic Claude, OpenAI GPT models, and Google Gemini, orchestrated via the Cline extension in Visual Studio Code—to construct a.NET MAUI test application for validating FinTech SDK wallet provisioning workflows on Android and iOS platforms.

- Recap of Primary Outcomes:

The study successfully demonstrated the generation of a functional.NET MAUI test application, albeit with a notable requirement for manual refinement and debugging of the AI-generated XAML and C# code. The AI models showed varying strengths: Claude models were adept at more complex C# logic and context retention, GPT-4 excelled at scaffolding UI and C# code-behind with precise prompts, and Gemini Pro was effective for smaller, well-defined tasks. The AI-generated test application, once refined, was capable of executing the defined wallet provisioning workflow against a mock FinTech SDK, with automated unit and UI tests confirming its operational correctness on both Android and iOS. The evaluation of AI-generated code quality revealed that while initial drafts could be produced rapidly, achieving MAUI-idiomatic, maintainable, and performant code required significant human expertise in prompt engineering and code review.

- Reinforcement of Important Findings:

The findings underscore the current viability of AI as an accelerator for generating initial code structures and boilerplate for.NET MAUI test applications, particularly in the FinTech domain where rapid testing of SDK integrations is crucial. However, the research also highlights critical limitations. The quality and usability of AI-generated code are profoundly dependent on the specificity, contextual richness, and iterative refinement of prompts. AI models still exhibit challenges in consistently adhering to framework-specific nuances of.NET MAUI, often requiring developers to possess deep framework knowledge to guide and correct the AI. While.NET MAUI itself provides a robust framework for cross-platform testing of FinTech SDKs, inherent complexities in mobile testing, such as UI consistency across platforms and efficient test script maintenance, persist even with AI assistance in app generation.

- Expert Views on Implications:

The outcomes of this research have several implications for mobile test automation engineers and the FinTech industry. For engineers, AI-assisted development can significantly speed up the initial creation of test environments and harnesses. However, this necessitates the acquisition of new skills in prompt engineering, AI model interaction, and critical validation of AI-generated artifacts. The role may shift from manual coding of test tools to orchestrating AI, designing effective prompts, and performing rigorous quality assurance on the AI's output. For the FinTech industry, the ability to more rapidly generate test applications for SDK integration testing could accelerate the adoption and deployment of new financial services. This is particularly relevant given the fast pace of innovation in FinTech. However, the reliance on AI also introduces caveats regarding the security and reliability of the generated test components themselves. Ensuring that AI-generated test code does not inadvertently introduce vulnerabilities or misinterpret critical SDK functionalities is paramount. Ultimately, AI holds the potential to democratize the creation of sophisticated test harnesses for complex SDKs, enabling more thorough and timely validation. Yet, this potential can only be realized if paired with expert human oversight, robust validation processes, and a continuous focus on the security and quality of both the AI-generated code and the FinTech services being tested. The journey towards a fully autonomous AI in this domain is still in its early stages, with the current paradigm being one of human-AI collaboration.

Acknowledgements

I want to extend my deepest gratitude to the vibrant and dedicated open-source .NET MAUI community. Their tireless work in developing and maintaining this powerful framework provided the essential foundation upon which this research was built. The collaborative spirit and shared knowledge within this community are truly inspiring and were instrumental in making this investigation possible.

Furthermore, I am profoundly grateful to the brilliant minds behind the AI models and tools leveraged in this research. Their groundbreaking contributions in artificial intelligence, ranging from advanced machine learning algorithms to robust development platforms, offered the critical analytical capabilities necessary for the validation processes explored in this article. Without these sophisticated technologies, the insights gained and the validation methodologies employed would not have been achievable.

REFERENCES

- OptiSol. (n.d.). Top 5 Reasons to Choose.NET MAUI for Cross-Platform Development. OptiSol Business Solutions. Retrieved from
- Leobit. (n.d.). Why is.NET MAUI an Excellent Cross-Platform Development Framework for Your Business? Leobit Blog. Retrieved from
- CorServ. (n.d.). 4 Reasons Why Fintechs Should Add Credit Cards to Their Services. CorServ Solutions. Retrieved from
- Patternica. (n.d.). Top Fintech API Platforms. Patternica Blog. Retrieved from
- Reddit user discussion. (n.d.). Tips for accelerating.NET MAUI app development. Reddit r/dotnetMAUI. Retrieved from
- MoldStud. (2023). Secrets to Successful.NET MAUI App Deployment in 2023: Expert Tips and Best Practices. MoldStud Blog. Retrieved from
- .NET Expert. (n.d.). Mastering UI Testing: A Comprehensive.NET MAUI Appium Tutorial..NET Expert Blog. Retrieved from
- Anthropic. (n.d.). Code with Claude. Anthropic Solutions. Retrieved from
- Anthropic. (n.d.). Claude Code Overview. Anthropic Documentation. Retrieved from
- Rizwan, S. (n.d.). Cline - Autonomous AI Coding Assistant. Visual Studio Marketplace. Retrieved from
- Syncfusion. (n.d.). New.NET MAUI AI AssistView Control for Building AI Chat Experience. Syncfusion Blogs. Retrieved from
- Syncfusion. (n.d.). .NET MAUI AI AssistView. Syncfusion. Retrieved from
- Apex Fintech Solutions. (n.d.). How a fintech with $123B of assets powers modern financial experiences through Speakeasy. Speakeasy Customer Stories. Retrieved from
- TestSigma. (n.d.). Fintech Application Testing: Ensuring Security and Reliability. TestSigma Blog. Retrieved from
- Split.io. (n.d.). SDK validation checklist. Split Help Center. Retrieved from
- Adobe. (n.d.). Validate your implementation. Adobe Experience Platform Mobile SDK Documentation. Retrieved from
- Keploy. (n.d.). AI Revolutionizes Software QA: Testing Frameworks. Keploy Blog. Retrieved from
- A Comprehensive Survey of AI-Driven Advancements and Techniques in Automated Program Repair and Code Generation. (2025, January 20). Aithor Paper Summary. Retrieved from
- Taylor, D. (n.d.). How to Hire.NET MAUI Developers with AI Usage. IT Supply Chain. Retrieved from
- Brainvire. (n.d.). .NET MAUI Beginner's Guide: Building Your First Cross-Platform App. Brainvire Blog. Retrieved from
- Belitsoft. (n.d.). Hire.NET MAUI Developers. Belitsoft. Retrieved from
- Microsoft. (n.d.). Manually provision an iOS app. Microsoft Learn. Retrieved from 80
- Runloop.ai. (n.d.). Assessing AI Code Quality: 10 Critical Dimensions for Evaluation. Runloop.ai Blog. Retrieved from 81
- Syncfusion. (n.d.). 10 Essential Prompt Engineering Criteria to Kickstart Your Success. Syncfusion Blogs. Retrieved from 70
- Belitsoft. (n.d.). System Prompt Engineering in Gen AI Applications. Belitsoft Blog. Retrieved from 77
- OpenAI. (n.d.). Six strategies for getting better results. OpenAI API Documentation. Retrieved from 75
- Anthropic. (n.d.). Claude 4 Best Practices for Prompt Engineering. Anthropic Documentation. Retrieved from 61
- Patil, P. (n.d.). Why Your.NET MAUI Mobile App Might Break Sooner Than You Think (And How to Fix It Early). Dev.to. Retrieved from
- AI for Automated Bug Detection and Debugging: A Comparative Study of Current Approaches. (2024). International Journal of Business Strategies and Management (IJBSM), 7(12). Retrieved from
- Geeta University. (n.d.). AI-Driven Software Testing & Bug Prediction: Revolutionizing the Future of Software Quality. Geeta University Blog. Retrieved from
- All Things Open. (n.d.). AI Code Assistants: Limitations to Consider. All Things Open Blog. Retrieved from
- Reddit user discussion. (n.d.). AI code suggestions sabotage software supply chain. Reddit r/programming. Retrieved from
- TestSigma. (2024, January 8). SDK Testing: A Comprehensive Guide. TestSigma Blog. Retrieved from
- Microsoft. (2024, June 6). Unit testing.NET MAUI apps. Microsoft Learn. Retrieved from
- Telerik. (2024, August 6). Integrating ChatGPT into.NET MAUI from Scratch. Telerik Blogs. Retrieved from
- Microsoft. (2024, August 30). .NET MAUI iOS entitlements. Microsoft Learn. Retrieved from
- BrowserStack. (2025, January 3). A Comprehensive Guide to SDK Testing. BrowserStack Guide. Retrieved from
- Microsoft. (2025, January 7). Improve.NET MAUI app performance. Microsoft Learn. Retrieved from
- Forbes Technology Council. (2025, January 24). Testing In Fintech: How Robust Testing Protects Your Apps And Customers. Forbes. Retrieved from
- Kumar, S. (2025, February 1). The Evolution of Software Testing: From Automation to AI. Aitude Blog. Retrieved from
- Leobit. (2025, February 20). .NET MAUI vs. Flutter: Comparison and Use Cases for the Most Popular Cross-Platform Development Frameworks. Leobit Blog. Retrieved from

- MoldStud. (2025, March 12). Exploring the Future of Mobile App Testing: Trends and Innovations You Need to Know. MoldStud Blog. Retrieved from
- Kathiresan, S. G. (2025, March 21). Creating a ChatGPT-Like App in.NET MAUI Using OpenAI API. Syncfusion Blogs. Retrieved from
- Kathiresan, S. G. (2025, March 21). Creating a ChatGPT-Like App in.NET MAUI Using OpenAI API (Full Post). Syncfusion Blogs. Retrieved from 65
- Heiskanen, M., et al. (2025, April 8). AI Adoption in Software Testing: Use Cases, Benefits, and Challenges in Industry. arXiv:2504.04921. Retrieved from
- Microsoft. (2025, April 9). Prompt engineering with.NET. Microsoft Learn. Retrieved from 72
- SDK.finance. (2025, April 10). The Fundamentals of FinTech Architecture: Trends, Challenges, and Solutions. SDK.finance Blog. Retrieved from
- Mia Platform. (2025, April 11). Software Development Lifecycle (SDLC) and AI: A New Era of Innovation. Mia Platform Blog. Retrieved from
- MetaDesign Solutions. (2025, April 14). How AI + Appium is Changing Mobile Testing in 2025. MetaDesign Solutions Blog. Retrieved from
- Microsoft. (2025, April 15). UI testing with Appium and.NET MAUI. Microsoft Learn. Retrieved from 79
- Innowise Group. (2025, April 25). .NET MAUI vs Xamarin: A Detailed Comparison for 2025. Innowise Blog. Retrieved from
- Google. (2025, May). Gemini API Models. Google AI for Developers. Retrieved from
- Microsoft. (2025, May 6). Tutorial: Create a.NET MAUI app. Microsoft Learn. Retrieved from 66
- Syncfusion. (2025, May 8). Top AI Code Editors Every Developer Should Know in 2025. Syncfusion Blogs. Retrieved from
- GitHub Copilot Team. (2025, May 8). Prompting GitHub Copilot Chat. Visual Studio Code Documentation. Retrieved from 71
- Microsoft. (2025, May 13). Evaluate LLM responses in.NET intelligent apps. Microsoft Learn. Retrieved from 82
- Microsoft. (2025, May 15). Evaluating AI content safety in.NET Intelligent Applications..NET Blog. Retrieved from 83
- Smart Scan: AI-Powered Code Analysis and Review. (2025, May 21). International Journal of Engineering Research & Technology (IJERT). Retrieved from
- Anthropic. (2025, May 22). Introducing Claude 4. Anthropic News. Retrieved from 62
- Blue Whale Apps. (2025, May 30). AI in Mobile App Technology 2025: Revolutionizing the Future of Apps. Blue Whale Apps Blog. Retrieved from 84

WORKS CITED

1. Top 5 Reasons to Choose .NET MAUI for Cross-Platform Development, accessed May 30, 2025, https://www.optisolbusiness.com/insight/top-5-reasons-to-choose-dot-net-maui-for-cross-platform-development
2. .NET MAUI vs. Flutter: Head-to-Head Comparison and Use Cases - Leobit, accessed May 30, 2025, https://leobit.com/blog/net-maui-vs-flutter-head-to-head-comparison-and-use-cases-for-the-most-popular-cross-platform-development-frameworks/
3. Xamarin vs .NET MAUI: differences, commonalities, which is best for ..., accessed May 30, 2025, https://innowise.com/blog/net-maui-vs-xamarin/
4. What is .NET MAUI Framework for Your Business? - Leobit, accessed May 30, 2025, https://leobit.com/blog/why-is-net-maui-an-excellent-cross-platform-development-framework-for-your-business/
5. Hire .NET Maui Developer in 2025 - Belitsoft. Software Development Company, accessed May 30, 2025, https://belitsoft.com/hire-maui-developers
6. Payment Card API Technology for Fintechs - CorServ Solutions, accessed May 30, 2025, https://www.corservsolutions.com/4-reasons-why-fintechs-should-add-credit-cards-to-their-services/
7. Top FinTech API Platforms for 2025 - Patternica, accessed May 30, 2025, https://patternica.com/blog/top-fintech-api-platforms
8. How a fintech with $123B of assets powers modern financial ..., accessed May 30, 2025, https://www.speakeasy.com/customers/apex
9. Fundamentals of FinTech Architecture: Challenges, and Solutions, accessed May 30, 2025, https://sdk.finance/the-fundamentals-of-fintech-architecture-trends-challenges-and-solutions/
10. Testing In FinTech: How Robust Testing Protects Your Apps And ..., accessed May 30, 2025, https://www.forbes.com/councils/forbestechcouncil/2025/01/24/testing-in-fintech-how-robust-testing-protects-your-apps-and-customers/
11. FinTech Application Testing To Achieve Fail-Proof Quality - Testsigma, accessed May 30, 2025, https://testsigma.com/blog/fintech-application-testing/
12. What is SDK Testing | BrowserStack, accessed May 30, 2025, https://www.browserstack.com/guide/sdk-testing
13. What is SDK Testing? How to Perform & Example - Testsigma, accessed May 30, 2025, https://testsigma.com/blog/sdk-testing/
14. Tips for Accelerating .NET MAUI App Development – Struggling with ..., accessed May 30, 2025, https://www.reddit.com/r/dotnetMAUI/comments/1izsfrb/tips_for_accelerating_net_maui_app_development/
15. Write beautiful code, ship powerful products | Claude by Anthropic ..., accessed May 30, 2025, https://www.anthropic.com/solutions/coding

16. Claude Code overview - Anthropic, accessed May 30, 2025, https://docs.anthropic.com/en/docs/claude-code/overview
17. Easily Build ChatGPT-like App in .NET MAUI using OpenAI APIs, accessed May 30, 2025, https://www.syncfusion.com/blogs/post/dotnet-maui-chatgpt-like-app-using-openai/amp
18. Integrating ChatGPT into .NET MAUI from Scratch - Telerik.com, accessed May 30, 2025, https://www.telerik.com/blogs/integrating-chatgpt-net-maui-scratch
19. Gemini models | Gemini API | Google AI for Developers, accessed May 30, 2025, https://ai.google.dev/gemini-api/docs/models
20. AI in Software Development | IBM, accessed May 30, 2025, https://www.ibm.com/think/topics/ai-in-software-development
21. Software Development Lifecycle (SDLC) and AI | Mia-Platform, accessed May 30, 2025, https://mia-platform.eu/blog/software-development-lifecycle-sdlc-and-ai/
22. Top AI Code Editors Every Developer Should Know in 2025 ..., accessed May 30, 2025, https://www.syncfusion.com/blogs/post/ai-code-editors-2025/amp
23. Smart-Scan: AI-Powered Code Analysis and Review – IJERT, accessed May 30, 2025, https://www.ijert.org/smart-scan-ai-powered-code-analysis-and-review
24. A Comprehensive Survey of AI-Driven Advancements and ... - Aithor, accessed May 30, 2025, https://aithor.com/paper-summary/a-comprehensive-survey-of-ai-driven-advancements-and-techniques-in-automated-program-repair-and-code-generation
25. Impressive Autonomous AI Code Generation with Cline, accessed May 30, 2025, https://chronicler.tech/impressive-autonomous-ai-code-generation-with-cline/
26. AI Coding Agent: With VSCode And Cline OSS - CodeSamplez.com, accessed May 30, 2025, https://codesamplez.com/productivity/ai-coding-agent
27. How to Hire .NET MAUI Developers with AI Usage - IT Supply Chain, accessed May 30, 2025, https://itsupplychain.com/how-to-hire-net-maui-developers-with-ai-usage/
28. FREE AI-Powered C# Code Generator: Use Context-Aware ... - Workik, accessed May 30, 2025, https://workik.com/c-sharp-code-generator
29. Cline - AI Autonomous Coding Agent for VS Code, accessed May 30, 2025, https://cline.bot/
30. Cline - Visual Studio Marketplace, accessed May 30, 2025, https://marketplace.visualstudio.com/items?itemName=saoudrizwan.claude-dev
31. AI code suggestions sabotage software supply chain : r/programming, accessed May 30, 2025, https://www.reddit.com/r/programming/comments/1jycix4/ai_code_suggestions_sabotage_software_supply_chain/
32. IDE integrations - Anthropic, accessed May 30, 2025, https://docs.anthropic.com/en/docs/claude-code/ide-integrations
33. How to optimally use Anthropic API through Cline in VS Code? : r ..., accessed May 30, 2025, https://www.reddit.com/r/ClaudeAI/comments/1i82dgg/how_to_optimally_use_anthropic_api_through_cline/
34. AI's Impact on Testing Frameworks & Software QA Evolution | Keploy ..., accessed May 30, 2025, https://keploy.io/blog/community/ai-revolutionizes-software-qa-testing-frameworks
35. The Evolution of Software Testing: From Automation to AI - AITUDE, accessed May 30, 2025, https://www.aitude.com/the-evolution-of-software-testing-from-automation-to-ai/
36. How to Use AI to Automate Testing—A Practical Guide (2025), accessed May 30, 2025, https://www.testdevlab.com/blog/how-to-use-ai-to-automate-testing
37. Using generative AI to create test cases for software requirements ..., accessed May 30, 2025, https://aws.amazon.com/blogs/industries/using-generative-ai-to-create-test-cases-for-software-requirements/
38. 5 Major Benefits Of Using AI in Software Testing | QA Training, accessed May 30, 2025, https://qatraininghub.com/5-major-benefits-of-using-ai-in-software-testing/
39. ijbemr.com, accessed May 30, 2025, https://ijbemr.com/wp-content/uploads/AI-FOR-AUTOMATED-BUG-DETECTION-AND-DEBUGGING-A-COMPARATIVE-STUDY-OF-CURRENT-APPROACHES.pdf
40. AI-driven Software Testing & Bug Prediction: Revolutionizing the ..., accessed May 30, 2025, https://blog.geetauniversity.edu.in/ai-driven-software-testing-bug-prediction-revolutionizing-the-future-of-software-quality/
41. How AI and Appium Are Revolutionizing Mobile Testing in 2025, accessed May 30, 2025, https://metadesignsolutions.com/how-ai-appium-is-changing-mobile-testing-in-2025/
42. The Future of Test Automation: Balancing Human Intelligence and AI ..., accessed May 30, 2025, https://www.lambdatest.com/blog/human-intelligence-and-ai-testing/
43. Future Trends and Innovations in Mobile App Testing | MoldStud, accessed May 30, 2025, https://moldstud.com/articles/p-exploring-the-future-of-mobile-app-testing-trends-and-innovations-you-need-to-know
44. 6 limitations of AI code assistants and why developers should be ..., accessed May 30, 2025, https://allthingsopen.org/articles/ai-code-assistants-limitations
45. arxiv.org, accessed May 30, 2025, https://arxiv.org/pdf/2504.04921

46. Unit testing - .NET MAUI | Microsoft Learn, accessed May 30, 2025, https://learn.microsoft.com/en-us/dotnet/maui/deployment/unit-testing?view=net-maui-9.0
47. Secrets to Successful NET MAUI App Deployment in 2023 - Expert Tips and Best Practices, accessed May 30, 2025, https://moldstud.com/articles/p-secrets-to-successful-net-maui-app-deployment-in-2023-expert-tips-and-best-practices
48. Improve app performance - .NET MAUI | Microsoft Learn, accessed May 30, 2025, https://learn.microsoft.com/en-us/dotnet/maui/deployment/performance?view=net-maui-9.0
49. Why Your .NET MAUI Mobile App Might Break Sooner Than You ..., accessed May 30, 2025, https://dev.to/p_0c0278d/why-your-net-maui-mobile-app-might-break-sooner-than-you-think-and-how-to-fix-it-early-288a
50. Mastering Automated UI Testing for .NET MAUI with Appium Tutorial, accessed May 30, 2025, https://dotnetexpert.net/blogs/mastering-ui-testing-a-comprehensive-.net-maui-appium-tutorial
51. SDK validation checklist – Split Help Center, accessed May 30, 2025, https://help.split.io/hc/en-us/articles/13998631077901-SDK-validation-checklist
52. Validating the Adobe Experience Platform Mobile SDK, accessed May 30, 2025, https://developer.adobe.com/client-sdks/home/getting-started/validate/
53. Introducing the New .NET MAUI AI AssistView Control - Syncfusion, accessed May 30, 2025, https://www.syncfusion.com/blogs/post/new-dotnet-maui-ai-assistview-control
54. .NET MAUI AI AssistView | Syncfusion, accessed May 30, 2025, https://www.syncfusion.com/maui-controls/maui-aiassistview
55. Build an AI-Powered Chat Experience with WinUI AI AssistView and ..., accessed May 30, 2025, https://www.syncfusion.com/blogs/post/ai-chat-with-winui-ai-assistview
56. Case Studies: Using Generative AI for Coding - Codecademy, accessed May 30, 2025, https://www.codecademy.com/resources/blog/case-studies-using-generative-ai-coding/
57. iOS entitlements - .NET MAUI | Microsoft Learn, accessed May 30, 2025, https://learn.microsoft.com/en-us/dotnet/maui/ios/entitlements?view=net-maui-9.0
58. Build your first .NET MAUI app - .NET MAUI | Microsoft Learn, accessed May 30, 2025, https://learn.microsoft.com/en-us/dotnet/maui/get-started/first-app?view=net-maui-9.0
59. .NET MAUI Beginner's Guide – Quick Introduction - Brainvire, accessed May 30, 2025, https://www.brainvire.com/blog/dotnet-maui-beginners-guide/
60. Install Visual Studio 2022 and Visual Studio Code to develop cross ..., accessed May 30, 2025, https://learn.microsoft.com/en-us/dotnet/maui/get-started/installation?view=net-maui-9.0
61. Claude 4 prompt engineering best practices - Anthropic, accessed May 30, 2025, https://docs.anthropic.com/en/docs/build-with-claude/prompt-engineering/claude-4-best-practices
62. Introducing Claude 4 - Anthropic, accessed May 30, 2025, https://www.anthropic.com/news/claude-4
63. Where can I find documentation for integrating GPT into a C# project? - API, accessed May 30, 2025, https://community.openai.com/t/where-can-i-find-documentation-for-integrating-gpt-into-a-c-project/585647
64. Integrating OpenAI's ChatGPT into cross-platform .NET applications, accessed May 30, 2025, https://platform.uno/blog/integrating-chatgpt-into-your-net-applications/
65. Easily Build ChatGPT-like App in .NET MAUI using OpenAI APIs - Syncfusion, accessed May 30, 2025, https://www.syncfusion.com/blogs/post/dotnet-maui-chatgpt-like-app-using-openai
66. Create a .NET MAUI app - .NET MAUI | Microsoft Learn, accessed May 30, 2025, https://learn.microsoft.com/en-us/dotnet/maui/tutorials/notes-app?view=net-maui-9.0
67. Code with Gemini Code Assist Standard and Enterprise | Gemini for Google Cloud, accessed May 30, 2025, https://cloud.google.com/gemini/docs/codeassist/write-code-gemini
68. Create prompts to generate code | Generative AI | Google Cloud, accessed May 30, 2025, https://cloud.google.com/vertex-ai/generative-ai/docs/code/code-generation-prompts
69. Introduction to prompt design | Gemini API | Google AI for Developers, accessed May 30, 2025, https://ai.google.dev/gemini-api/docs/prompting-intro
70. 10 Essential Prompt Engineering Criteria to Kickstart Your Success, accessed May 30, 2025, https://www.syncfusion.com/blogs/post/10-prompt-engineering-criteria
71. Prompt engineering for Copilot Chat - Visual Studio Code, accessed May 30, 2025, https://code.visualstudio.com/docs/copilot/chat/prompt-crafting
72. Prompt Engineering concepts - .NET | Microsoft Learn, accessed May 30, 2025, https://learn.microsoft.com/en-us/dotnet/ai/conceptual/prompt-engineering-dotnet
73. Prompt Engineering Best Practices: Tips, Tricks, and Tools | DigitalOcean, accessed May 30, 2025, https://www.digitalocean.com/resources/articles/prompt-engineering-best-practices
74. Prompt Engineering for AI Guide | Google Cloud, accessed May 30, 2025, https://cloud.google.com/discover/what-is-prompt-

engineering
75. Prompt engineering - OpenAI API - OpenAI Platform, accessed May 30, 2025, https://platform.openai.com/docs/guides/prompt-engineering/six-strategies-for-getting-better-results
76. accessed December 31, 1969, https://www.syncfusion.com/blogs/post/10-essential-prompt-engineering-criteria-to-kickstart-your-success
77. System Prompt Engineering in Gen AI Applications | Belitsoft, accessed May 30, 2025, https://belitsoft.com/system-prompt-engineering-in-gen-ai-applications
78. Advanced Prompt Engineering Techniques - Mercity AI, accessed May 30, 2025, https://www.mercity.ai/blog-post/advanced-prompt-engineering-techniques
79. .NET MAUI - UI testing with Appium and NUnit - Code Samples ..., accessed May 30, 2025, https://learn.microsoft.com/en-us/samples/dotnet/maui-samples/uitest-appium-nunit/
80. Manual provisioning for iOS apps - NET MAUI - Learn Microsoft, accessed May 30, 2025, https://learn.microsoft.com/en-us/dotnet/maui/ios/device-provisioning/manual-provisioning?view=net-maui-9.0
81. Assessing AI Code Quality: 10 Critical Dimensions for Evaluation ..., accessed May 30, 2025, https://www.runloop.ai/blog/assessing-ai-code-quality-10-critical-dimensions-for-evaluation
82. The Microsoft.Extensions.AI.Evaluation libraries - .NET | Microsoft ..., accessed May 30, 2025, https://learn.microsoft.com/en-us/dotnet/ai/conceptual/evaluation-libraries
83. Evaluating content safety in your .NET AI applications - .NET Blog, accessed May 30, 2025, https://devblogs.microsoft.com/dotnet/evaluating-ai-content-safety/
84. AI in Mobile App Technology 2025: Revolutionizing the Future of Apps, accessed May 30, 2025, https://bluewhaleapps.com/blog/ai-in-mobile-app-technology-2025