JOURNAL OF INFORMATION SCIENCE AND ENGINEERING XX, XXX-XXX (2026)
DOI: 10.6688/JISE.20XX0X_XX(X).00XX

An Open Super-App Construction Method

FU-CHIUNG CHENG
Department of Computer Science and Engineering
Tatung University
Taipei, 104 Taiwan

E-mail: fccheng@gm.ttu.edu.tw

Super apps consolidate diverse services into a single container, but composing multiple applications into a coherent super app remains challenging. Dominant ecosystems are generally closed and platform-controlled, limiting architectural transparency and confining developers to predefined extension models. Consequently, organizations often cannot design and govern their own super apps, particularly their interoperability, security boundaries, and lifecycle management, and must rely on reimplementation, brittle interfaces, or heavyweight frameworks that increase security risks and maintenance costs. We propose an open, modular, tool-assisted method that builds a super app by composing a MainApp with one or more independently operable application modules. Interoperability is provided by two MainApp components: an Information Transfer Manager (ITM) for controlled cross-module data exchange and a Background API Executor (BAE) for invoking registered module background entry points. The method also supports hierarchical composition by embedding one super app within another under a single governance plane. A preliminary quantitative case-study evaluation examines runtime resource footprint, page-load behavior, and end-to-end composition-path latency. The measurements indicate a combined application-specific footprint of 950.60 KB and an average composition-path latency of 602.4 ms. These results are descriptive and do not replace a statistically powered cross-platform benchmark. Overall, the proposed method preserves application reuse and enables developer-governed super apps outside closed ecosystems.

Keywords: *Super App Construction, Open Super-App Architecture, Modular Composition, Information Transfer Manager, Background API Executor, Performance Evaluation*

1. INTRODUCTION

Super apps aim to deliver a one-stop solution by integrating communication, commerce, payments, travel, and other services into a single product surface that operates as a unified in-app ecosystem rather than as a collection of standalone applications. This configuration can reshape distribution, competition, and innovation by shifting user journeys from app stores to platform-centric ecosystems [1].

Although the super app phenomenon is most strongly associated with major Asian platforms (e.g., LINE [2], WeChat [3] and Grab [4]), research also reports growing interest beyond Asia, including adoption evidence in Europe and broader user-experience analyses [1, 5–6]. However, dominant super-app ecosystems are typically closed and platform-controlled: third-party developers are limited to predefined extension mechanisms and lack access to the core orchestration plane. Consequently, most organizations have no practical way to design, deploy, and govern their own super app, including its interoperability model, security boundaries, and lifecycle management.

This work describes and formalizes a method for constructing a super app by selecting a MainApp and one or more independently operable application modules, and then coordinating cross-module data transfer and background execution through the MainApp. This work makes the following contributions:

1. **A concrete super-app composition method** with a stepwise workflow: selecting a MainApp, instantiating shared orchestration components, selecting application modules, constructing transmission paths, and binding background execution entry points.
2. **A reference architecture** that separates the startup file, program logic master file, and background API master file, enabling non-UI operations to execute without blocking the main interface.
3. **Hierarchical super-app composition** that disables an imported super-app's internal orchestration components and re-registers its module paths, namespaces, and background entry points under the outer MainApp.

The remainder of this paper is organized as follows. Section 2 reviews representative super-app ecosystems and existing development approaches. Section 3 presents the proposed architecture, composition algorithm, runtime governance mechanisms, and reference implementation. Section 4 reports preliminary quantitative case-study measurements. Section 5 discusses security, adoption, user-experience governance, and socio-organizational implications. Section 6 concludes the paper and outlines future work.

2. BACKGROUND AND RELATED WORK

Super apps are commonly viewed as platform ecosystems rather than feature-rich applications. Mini-app ecosystems consist of lightweight services that run inside a host application, reducing installation friction while expanding platform scope. WeChat Mini Programs are a representative example of this in-app service model [3].

A mini-app typically targets a single, well-defined use case, whereas a super app aggregates multiple related services within one user-facing container. Compared with traditional apps, super apps provide shared service integration, unified identity, habitual multi-touchpoint use, and an ecosystem-oriented expansion model. Table 1 summarizes these distinctions as an author synthesis based on prior super-app research [1, 6]. The following subsections examine three representative super apps: LINE, WeChat, and Grab.

Table 1. Differences between traditional apps and super apps, based on [1, 6].

Aspect	Traditional app	Super app
Core purpose	Solves a single, defined use case	Solves a single, defined use case Supports multiple related use cases
Service integration	Limited or none	Shared integration across services
User identity	App-specific	Unified identity across services
Typical usage pattern	Task-based, episodic	Habitual, multi-touchpoint
Ecosystem model	Standalone product	Platform or ecosystem
Expansion approach	Feature additions	Adjacent services layered onto a core
Monetization model	Single revenue stream	Multiple, interconnected revenue streams

2.1 LINE as a Representative Super App

LINE evolved from a messaging application into a multi-service ecosystem that includes official accounts and embedded web experiences such as LINE MINI App. Its extension interfaces illustrate the convenience and modular expansion associated with host-controlled super-app platforms [2].

However, third-party developers operate through prescribed interfaces and remain dependent on platform policies, capability boundaries, and review processes. LINE therefore demonstrates the value of a unified service ecosystem while motivating developer-owned orchestration and explicit cross-module interoperability [2].

2.2 WeChat as a Representative Super App

WeChat is a canonical super app that combines communication, payments, content, commerce, and Mini Programs within a single host environment [3]. Its scale demonstrates the benefits of modular services and reduced app switching, but also increases governance and security demands.

Research identifies identity-confusion risks in app-in-app ecosystems and AppSecret leakage in WeChat Mini Programs, highlighting the need for rigorous permission checks, credential protection, and auditable data flows [7–8]. These limitations motivate a developer-governed orchestration boundary for controlled data exchange and background execution.

2.3 Grab as a Representative Super App

Grab evolved from a mobility service into a Southeast Asian multi-service platform that integrates transport, delivery, and financial services [4]. It illustrates how a high-frequency core service can expand through shared platform capabilities and a consistent user experience. From the architectural perspective used in this paper, Grab remains a host-controlled ecosystem: third parties do not own the platform's identity, navigation, or orchestration plane.

2.4 Existing Super-App Development Approaches and Tools

Existing super-app construction solutions can be grouped into four broad approaches. The first is the **host-controlled mini-app model** used by ecosystems such as WeChat and LINE. It provides a mature runtime, shared identity, distribution, and platform APIs, but applications must conform to proprietary manifests, review rules, capability boundaries, and lifecycle policies. This approach is effective for extending an established host, but it does not allow an organization to own the complete orchestration plane or freely compose independently deployable applications into a new super app [2–3].

The second approach is **micro-frontend composition**. Micro-frontends decompose a user interface into independently developed and deployable units and commonly use build-time integration, runtime JavaScript composition, Web Components, iframes, or module federation. The approach improves team autonomy and incremental evolution, but can introduce dependency duplication, inconsistent user experience, routing complexity, shared-state coordination, and runtime isolation concerns. Its use in native mobile environments is more constrained because packaging, navigation, and platform lifecycle behavior must be coordinated explicitly [9–10].

The third approach is **native modularization**. Android dynamic features and iOS frameworks can separate application functions into modules and support deferred delivery or controlled extension. These mechanisms provide strong platform integration and performance, but they are operating-system specific, require substantial engineering effort, and generally do not define a cross-platform super-app governance model for data exchange, background invocation, and nested composition [11–12].

The fourth approach uses **cross-platform and low-code development tools**. Frameworks and commercial platforms reduce duplicated implementation across devices and accelerate UI and workflow development. However, their primary abstraction is usually a single application project or a vendor-managed component model; they do not, by themselves, specify how multiple pre-existing applications remain independently operable under a developer-owned orchestration boundary [13–14].

A comparison of representative super-app development approaches and the proposed method, which is detailed in Section 3, is summarized in Table 2.

Table 2. Comparison of representative super app development approaches.

Approach	Unit	Control	Interoperability	Primary limitation
Host mini-app	Mini/partner app	Platform owner	Host APIs/services	Closed governance; host dependency
Micro-frontend	UI fragment	App shell	Events/state/routing	Runtime and UX consistency
Native	OS module	Native host	Custom contracts	Platform-specific; high effort
Cross-platform/ low-code	Component/ project	Framework/ vendor	Framework/ custom APIs	Usually single-project or vendor-centric
Proposed method	Standalone app	Developer MainApp	ITM + registered BAE	Requires explicit contracts / policies

2.5 Research Gap and Positioning of the Proposed Method

Existing approaches leave a gap between closed mini-app ecosystems and general modular application architectures. Host platforms provide integrated services but retain control over orchestration and governance, whereas micro-frontends and native architectures do not offer a unified cross-platform model for composing standalone applications, governing data exchange, executing background functions, and supporting nested super apps.

The proposed method addresses this gap through application-level composition, preservation of standalone operability, a developer-controlled MainApp as the orchestration boundary, and explicit interoperability through the Information Transfer Manager and Background API Executor. MobiWebX [15–16] is used only as a reference implementation; the primary contribution is the platform-independent architecture, contracts, and runtime model presented in Section 3.

3. PROPOSED OPEN SUPER-APP ARCHITECTURE

This section formalizes the proposed architecture, module contracts, interoperability mechanisms, composition algorithm, and runtime semantics. MobiWebX is used only as a reference implementation to demonstrate feasibility; IDE-specific operating instructions are intentionally omitted.

3.1 Super App System Model

Super apps integrate diverse services into a single product surface to deliver a one-stop user experience. From a system design perspective, the core objectives are **multi-function integration, modularity, unified UX**, and **high extensibility**.

Our model achieves these objectives by composing independently developed application projects under a single entry point (the **MainApp**) while keeping each module separable for maintenance and future expansion. The design goals are:

- **Low-friction reuse:** import existing app projects as composition units with minimal refactoring.
- **Interoperability:** enable cross-module data exchange through a systematic mechanism.
- **Background orchestration:** allow the MainApp to trigger application modules' background functionality when needed.
- **Compositional scalability:** support importing one super app into another while preserving governance clarity.

A central assumption is independent operability: every app integrated as a module must run correctly as a standalone project before composition. This constraint prevents hidden cross-module dependencies and ensures that assembly remains primarily a configuration and orchestration task rather than a refactoring exercise.

Architecture Overview

Fig. 1 illustrates the proposed architecture, in which a MainApp serves as the initial entry point and orchestration boundary for one or more independently operable application modules. These modules provide services such as communication, e-commerce, travel, and payments while preserving their internal logic and remaining deployable as standalone Android, iOS, or web applications.

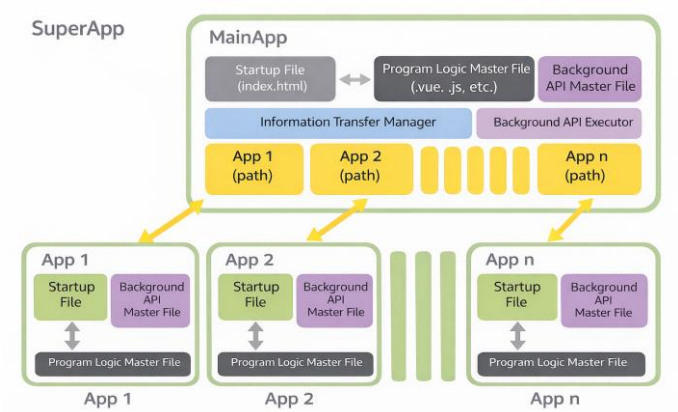


Fig. 1. Proposed Super-App Architecture.

The MainApp comprises three implementation files—a startup file, program logic master file, and background API master file—and two runtime components: the Information Transfer Manager and Background API Executor. The startup file initializes the runtime and loads required resources, while the program logic master file defines the application pages, UI components, state, and executable functions, including API calls. The Information Transfer Manager governs data exchanged among applications in the super app and enables authorized modules to read shared data through a common interface. A sender may encrypt sensitive data for decryption by an authorized receiver. The background API master file declares the module functions that may be invoked asynchronously through the Background API Executor.

Through these mechanisms, multiple applications can be combined into a unified super app experience while preserving modularity and reducing the need for refactoring.

Each application module in the proposed super-app architecture comprises three primary artifacts: **a startup file**, **an optional background API master file**, and **a program logic master file**. These artifacts serve functions analogous to the corresponding MainApp artifacts. Each module remains independently operable and can be compiled as an Android application, an iOS application, or a website.

Super-App Runtime

At runtime, the MainApp initializes the composition manifest, the Information Transfer Manager, and the Background API Executor. The composition manifest maintains module identifiers, startup paths, declared data capabilities, and registered background entry points. The Information Transfer Manager mediates governed data exchange, whereas the Background API Executor dispatches authorized background operations.

Information Transfer Manager (ITM)

The ITM is a MainApp-resident component that provides controlled and auditable data exchange among application modules. It replaces ad hoc mechanisms, such as unmanaged localStorage keys, with a governed data plane based on namespaces, schemas, access policies, metadata, and lifecycle rules. The ITM operates as follows:

1. **Producer writes:** an application module publishes data via ITM.put(namespace, key, payload, metadata).
2. **Policy enforcement: ITM authenticates the caller, validates the payload schema, and enforces read/write permissions, size limits, and sensitivity-classification policies.**
3. **Store and assign metadata:** ITM stores the payload with metadata (timestamp, TTL, owner, version, sequence, and policy).
4. **Consumer reads:** another module retrieves data via ITM.get(namespace, key).
5. **Lifecycle management: ITM expires or deletes records, increments versions, resolves conflicts, and optionally notifies subscribed modules according to policy.**

The ITM provides four core functions: (A) it standardizes cross-module data exchange through a single interface for publishing and consuming shared data (key–value pairs or structured objects) without tight coupling; (B) it enforces governance by controlling who may read/write, which fields are permitted, data expiration/retention rules, and conflict-resolution policies; (C) it supports common interoperability patterns, including

parameter passing (inputs to the next module), shared global state (e.g., user profile, session context, preferences), and event-like handoffs (publishing a "completed/updated" payload to trigger subsequent steps); and (D) it can optionally audit and log transfers for debugging, compliance, and incident response.

ITM Conflict-Resolution Strategy

Each shared item is represented as a *record R* = ***<namespace, key, value, version, sequence, timestamp, owner, policy>***. The default ITM strategy uses optimistic concurrency control when a writer supplies an expected version. The ITM accepts the write only when the expected version equals the current record version, increments the version and MainApp-controlled sequence atomically, and returns the updated record. For keys explicitly configured to permit blind writes, the ITM applies last-writer-wins using the tuple *<version, sequence, timestamp>*, in which version and sequence provide deterministic ordering and timestamp supports auditing. A rejected writer receives a structured conflict result containing the current version and value and may retry, merge, or abort.

The default policy can be overridden per namespace or key. Four policies are supported: (1) reject-on-conflict for security-sensitive state such as authentication or payment status; (2) Last-writer-wins for replaceable preferences, such as language or theme, where the most recently accepted value supersedes the previous value; (3) field-level merge for structured objects whose fields are independently owned; and (4) append-only event accumulation for logs, notifications, or workflow events. For commutative numeric states, an application-defined reducer, such as increment, minimum, or maximum, may be registered.

All policy selections are declared in the composition manifest, and each conflict decision is logged. For example, if two modules read version 12 of global.userProfile, the first accepted update creates version 13. A later write still based on version 12 is rejected, preventing silent overwriting and enabling deterministic, auditable recovery.

Background API Executor (BAE)

The BAE is a MainApp-resident orchestration component that invokes registered module functions without switching the active UI. It supports background tasks such as data retrieval, computation, synchronization, logging, and workflow execution. The BAE execution flow is:

1. **Trigger:** The MainApp submits a request in response to a user action, timer, lifecycle event, or module completion.
2. **Authorize:** The BAE validates the caller, target module, entry point, and permitted data access.
3. **Bind inputs:** Bounded parameters are passed to the selected entry point.
4. **Execute**: The function runs in a background context and returns a result object.
5. **Return and chain**: The MainApp receives a structured success or failure response, updates its state, and may initiate subsequent actions.

The BAE therefore provides three capabilities: authorized dispatch to registered background entry points, asynchronous execution outside the UI thread, and typed results that support notifications, state updates, and deterministic workflow chaining.

Nested Super-App Composition

The proposed method supports hierarchical composition of multiple super apps:

- **Single-governor rule:** When a super app is imported into another MainApp, it is treated as a standard application module. Its internal ITM and BAE are disabled, leaving the parent MainApp's ITM and BAE as the sole control plane.
- **Concrete effect:** When super app B is imported into super app A, B's MainApp and constituent modules are registered as application modules of A, with their directory paths generated accordingly.

This single-governor model eliminates competing control planes and centralizes policy enforcement for cross-module data exchange and background execution. During nested import, the outer MainApp assigns each constituent module a globally unique identifier and namespace, such as parentSuperApp.moduleId.namespace. Duplicate identifiers, entry-point names, or incompatible policies must be rejected or explicitly remapped before composition.

UI Consistency and Style Isolation

The MainApp defines a shared UI contract covering design tokens, navigation, typography, spacing, color roles, accessibility, and theme versioning. Application modules consume these settings through CSS custom properties or an equivalent theme API, allowing incompatible modules to be rejected or adapted during import.

To prevent mutual CSS interference, a conforming implementation uses one of three isolation levels. The preferred level is Shadow DOM or an equivalent encapsulated component boundary, which prevents selectors from crossing module boundaries while allowing approved design tokens to inherit. Where Shadow DOM is unavailable, all selectors are automatically prefixed with a module-specific root namespace, global selectors such as body, html, *, and unscoped element rules are rejected, and a build-time linter verifies that styles remain within the module root. For untrusted third-party modules, iframe/WebView isolation may be used, with communication restricted to the ITM and BAE interfaces.

The MainApp also governs shared overlays, dialogs, navigation bars, and z-index ranges, enabling modules to retain independent layouts while preserving a consistent user experience and avoiding style or event-handler conflicts.

3.2 Architecture Realization and Composition Algorithm

The proposed method is platform-independent at the architectural level. A conforming implementation must provide: (1) a main application that owns the composition manifest and orchestration policies; (2) independently executable modules with defined startup and optional background entry points; (3) an information-transfer service that mediates cross-module state; and (4) a background executor that dispatches authorized calls and returns structured results.

Given independently operable applications $A = \{a_0, a_1, ..., a_n\}$, the construction procedure selects a_0 as MainApp, assigns each a_i $(i > 0)$ a unique module identifier and startup path. The MainApp then registers each module's data capabilities, background entry points, and policy bindings. Composition succeeds only after standalone execution, iden-

tifier uniqueness, entry-point availability, and policy validation are confirmed. At runtime, the MainApp resolves navigation, routes data through the ITM, and dispatches non-UI operations through the BAE.

For nested composition, the imported super app's internal orchestration components are disabled, and its constituent modules are re-registered under the outer MainApp. Qualified identifiers and namespaces preserve a single control plane and prevent conflicts among module IDs, ITM keys, background entry points, and policies.

3.3 Reference Implementation and Case-Study Validation

A MobiWebX [15–16] reference implementation was developed with one MainApp, **SuperApp_Home**, and four independently executable modules: **SuperApp_BMI**, **SuperApp_AIDoctor**, **SuperApp_StoreFront**, and **SuperApp_StoreManager**. These modules represent numerical computation, AI-service invocation, customer-facing e-commerce, and administrative management. Each module was verified as a standalone application before being registered and imported through the composition manifest.

The case study demonstrated **deterministic module launching**, **cross-module data transfer**, and **centralized orchestration**. In particular, the BMI result generated by SuperApp_BMI was transferred to SuperApp_AIDoctor and incorporated into an AI-assisted recommendation request. The MainApp coordinated navigation, data exchange, and background invocation without duplicating module logic, demonstrating the functional feasibility of the proposed composition model.

The prototype uses localStorage and application-level background instrumentation; therefore, the measurements in Section 4 represent storage-write, storage-read, and dispatch paths rather than complete production ITM and BAE overhead. A production implementation should replace unmanaged storage access with governed ITM interfaces and expose background functions only through authorized, resource-bounded BAE entry points. MobiWebX is described in [15], and the detailed construction procedure appears in Chapter 8 of [16]; the primary contribution remains the platform-independent architecture and runtime governance model.

4. PRELIMINARY PERFORMANCE EVALUATION

Because the reference implementation provides limited application-level instrumentation, this section reports a preliminary case study rather than a complete multi-platform benchmark. The evaluation covers runtime resource footprint, page-load performance, and end-to-end latency of the BMI-to-AI-Doctor workflow. Page-load tests used five runs per page with a 30-second content-detection timeout, and the workflow test used five runs. Because device, browser or WebView, cache, and network conditions were not recorded, the results are descriptive and not independently reproducible.

Runtime Resource-Footprint Evaluation

To quantify the storage and loading footprint of the reference implementation, the MainApp and four imported modules were measured after execution. The measurement distinguishes among: (1) static incremental resources associated with each application page; (2) runtime as-

sets discovered from image and background-resource URLs after execution; (3) corrected incremental size, obtained by combining the first two components; (4) full declared page resources; and (5) full runtime footprint, defined as full declared resources plus runtime assets. Full runtime size is the total resource footprint reported after execution, including declared and runtime-discovered resources after accounting for overlapping or deduplicated assets. All values except Runtime assets in Table 3 are expressed in KB.

Table 3. Runtime resource footprint of the reference super-app implementation.

Application module	Static incremental (KB)	Runtime assets (count)	Runtime asset size (KB)	Corrected incremental size (KB)	Shared framework size (KB)	Full declared (KB)	Full runtime size (KB)
Home	14.68	4	213.10	227.78	N/A	9,517	9,725
BMI	13.49	0	0.00	13.49	8,710	9,514	9,514
AI Doctor	13.62	0	0.00	13.62	8,710	9,514	9,514
Store Front	9.33	14	673.02	682.34	8,710	8,719	9,376
Store Manager	8.85	4	4.51	13.36	8,710	8,718	8,723
Total application-specific resources	59.96	22	890.63	950.59	—	—	—

Note: Corrected incremental size is estimated independently from full declared size. The categories are not mutually exhaustive and should not be added to reconstruct the full declared resource size.

The combined application-specific footprint was 950.59 KB, comprising 59.96 KB of static resources and 890.63 KB of runtime assets. Thus, most of the incremental footprint resulted from application content rather than the super-app composition and orchestration mechanisms.

The BMI, AI Doctor, and Store Manager modules have corrected incremental footprints of only 13.36–13.62 KB, representing approximately 0.14%–0.15% of their 8,723–9,514 KB full runtime footprints. This suggests that module-specific resources are small relative to the common framework, although physical deduplication across deployments was not verified. The Home entry is marked N/A for shared-framework size because it serves as the baseline container.

Store Front has the largest corrected incremental footprint at 682.34 KB, of which 673.02 KB comes from 14 runtime assets, including a 418.92 KB product image. Its larger size therefore results mainly from media content rather than the composition architecture. The MainApp contributes 227.78 KB, including 213.10 KB for four module-selection icons and only 14.68 KB of static resources. This finding suggests that the structural resources needed to implement the MainApp, module registry, and launch interface are relatively small. Most of the MainApp increment is determined by user-interface assets rather than by orchestration logic.

Overall, the measurements are consistent with relatively small application-specific resource increments for modules without substantial media content. Such modules introduce only about 13–14 KB beyond the common framework in the tested implementation.

Page-Load Performance Evaluation

Five page-load measurements were performed for the MainApp and each of its four modules. Each run began at navigation initiation and ended when the expected page-specific text appeared, with no additional fixed delay. Three metrics were recorded: Goto Return, the time for the navigation command to return; DOM Ready, the time until the target document reached the ready state; and Content Detect, the time until the expected content appeared. Table 4 reports the results in milliseconds. Avg., P50, and Min–Max denote the mean, median, and observed range; Successful avg. includes only successful detections, and Timeouts denotes timed-out runs.

Table 4. Page-load performance of the MainApp and four application modules.

Page	Goto average (ms)	DOM Ready Avg./P50/Min-Max (ms)	Content Detect Avg./P50/ Min-Max (ms)	Succesful-run mean (ms)	Timeouts (count)	Success Rate (%)
Home	16.4	28.6 / 30 / 24-32	43.8 / 44 / 40-47	43.8	0	100%
BMI	42.0	122.4 / 32 / 26-481	140.6 / 49 / 38-507	140.6	0	100%
AI Doctor	18.6	45.2 / 33 / 24-109	61.8 / 48 / 36-129	61.8	0	100%
Store Front	21.4	61.4 / 35 / 27-174	12,075.6/205/50-30,040	103.3	2	60%
Store Manager	18.2	55.2 / 28 / 24-164	70.8 / 39 / 34-193	70.8	0	100%

The MainApp loaded consistently, with an average DOM Ready time of 28.6 ms and an average Content Detect time of 43.8 ms; the latter ranged from 40 to 47 ms. The BMI Calculator had a higher average Content Detect time of 140.6 ms because its first run required 507 ms. The remaining four runs averaged 49.0 ms, suggesting a possible first-run or cache-initialization effect. AI Doctor and Store Manager showed similar behavior, with average Content Detect times of 61.8 ms and 70.8 ms and median values of 48 ms and 39 ms, respectively.

Store Front requires separate interpretation. Although its median DOM Ready time was 35 ms, two of five runs timed out because the expected text, “GreenLife Supply,” did not appear within 30 seconds; the login page appeared instead. Including timeout-capped values produced a Content Detect mean of 12,075.6 ms, which should not be treated as a normal page-load average. The three successful runs averaged 103.3 ms.

Overall, 23 of 25 runs successfully displayed the expected content, yielding a 92% success rate and a pooled successful-run Content Detect mean of 82.4 ms. Four of the five pages had median Content Detect times between 39 and 49 ms. These results indicate low loading latency for most modules in the tested environment, but they remain preliminary because only five runs per page were performed in a single uncontrolled environment. Future evaluation should include more repetitions, controlled cache and network conditions, multiple devices and browsers, and larger module sets.

End-to-End BMI-to-AI-Doctor Workflow Latency

The BMI-to-AI-Doctor workflow was executed five times. Each run included BMI computation, cross-module storage writing, navigation to the AI Doctor module, storage reading, background dispatch, and an external AI-service request. Because the prototype uses localStorage and application-level dispatch, the measurements represent the implemented composition path rather than complete production ITM and BAE overhead. The composition-path latency is

defined as:

$$T_{\text{composition}} = T_{\text{storage write}} + T_{\text{navigation}} + T_{\text{storage read}} + T_{\text{BG dispatch}}$$

The complete workflow latency is defined as:

$$T_{\text{complete}} = T_{\text{BMI}} + T_{\text{composition}} + T_{\text{external service}}$$

Table 5 reports the measured components in milliseconds. The external-service value represents time to a response or error; because most requests returned HTTP 403, it must not be interpreted as successful AI-inference latency.

Table 5. Latency of the BMI-to-AI-Doctor workflow over five runs.

Run	BMI Computation	Storage write	Module Navigation	Storage read	BG dispatch	External-service response/error	Composition path $T_{\text{composition}}$ (ms)	Complete workflow T_{complete} (ms)
1	13	361	216	35	21	279	633	925
2	12	316	216	24	17	481	573	1,066
3	12	358	216	15	13	323	602	937
4	12	347	216	14	14	279	591	882
5	14	365	216	20	12	309	613	936
Average	12.6	349.4	216.0	21.6	15.4	334.2	602.4	949.2
Median	12	358	216	20	14	309	602	936
Min./max.	12/14	316/365	216/216	14/35	12/21	279/481	573/633	882/1,066

The mean composition-path latency was 602.4 ms, ranging from 573 to 633 ms. The complete workflow averaged 949.2 ms, with a median of 936 ms. Four of five runs returned a response or error within one second. Run 4 was the shortest at 882 ms, whereas Run 2 required 1,066 ms because its external-service response/error time increased to 481 ms.

The instrumented storage-write path was the largest component, averaging 349.4 ms and accounting for 58.0% of $T_{\text{composition}}$. Navigation contributed 216.0 ms (35.9%), while storage reading and background dispatch required 21.6 ms and 15.4 ms, respectively. Because the storage-write value includes the application-level instrumented path, it should not be interpreted as the intrinsic latency of localStorage.setItem(). These results identify prototype-level optimization targets rather than production ITM or BAE overhead.

The composition path represented 63.5% of the complete workflow latency, external-service communication 35.2%, and BMI computation 1.3%. BMI computation remained between 12 and 14 ms. The composition-path latency had a sample standard deviation of 22.6 ms and a coefficient of variation of 3.75%; however, these values underestimate variability because the same 216-ms navigation measurement was reused in all five runs.

Overall, the workflow reached an external-service response or error in approximately one second. Because the experiment included only five runs, reused navigation measurements, and mostly HTTP 403 responses, the findings are descriptive rather than generalizable. Future evaluation should use at least 30 independent runs, successful authenticated requests, separate cold- and warm-start tests, per-run navigation measurements, and direct instrumentation of the production ITM and BAE paths.

5. DISCUSSION

Super apps consolidate identity, payments, data, and third-party services within a unified interface and governance structure. Their convenience and scalability are accompanied by increased architectural complexity, security risk, and socio-technical impact.

5.1 Security Considerations

Super-app composition creates trust boundaries because modules may be developed by different parties. The MainApp should therefore treat imported modules as untrusted by default and enforce least-privilege access. Module provenance and integrity should be verified through mechanisms such as code signing and trusted-developer allowlists. Each module should declare its required capabilities in a manifest. Cross-module data exchange should be mediated by the ITM, which enforces namespaces, schema validation, size limits, retention rules, and optional audit logging. Background execution should likewise be isolated and resource-bounded. The BAE should invoke only registered entry points, enforce time and memory limits, and return structured results to prevent faulty modules from degrading UI responsiveness or system stability. These controls address reported identity-confusion and credential-leakage risks in existing app-in-app ecosystems [7–8].

5.2 Adoption is driven by value, constrained by risk and trust

Adoption is driven primarily by perceived utility and economic value, while perceived risk remains a barrier, particularly in Western markets [5]. Indonesian digital-banking evidence further highlights habit and enabling conditions as determinants of continued use [17]. Satisfaction, engagement, service quality, and convenience also shape durable adoption and loyalty [1, 18].

5.3 UX scaling demands coherence and deliberate feature governance

Super apps evolve into daily-life ecosystems whose interfaces must remain coherent across heterogeneous services [6]. Requirements-engineering research therefore treats service selection as an optimization problem rather than unrestricted feature accumulation [20]. Shared design tokens, navigation primitives, and shortcut mechanisms can reduce interaction debt as the ecosystem expands [6, 21].

5.4 Socio-organizational externalities are first-order design constraints

Super apps can reshape communication norms and intensify work–life boundary pressures; users respond through delayed replies, account separation, and muting [22]. Enterprise deployments may also create overload, invasion, and uncertainty that reduce employee engagement [23]. Native quiet hours, role-based profiles, policy-aligned defaults, training, and change-management measures should therefore be treated as first-order governance requirements.

6. CONCLUSIONS

This paper presented an open super-app construction method that composes independently operable applications under a developer-owned MainApp. Unlike closed mini-

app ecosystems, the method gives organizations control over orchestration and governance. It treats complete applications as reusable modules and provides explicit mechanisms for governed data exchange, background execution, deterministic startup-path addressing, and nested composition under a single control plane. A four-module reference implementation demonstrated module reuse, deterministic launch, and cross-module data transfer.

Preliminary measurements examined application-specific resource footprint, page-load behavior, and cross-module workflow latency. Although the results support the feasibility of the reference implementation, they do not validate production ITM and BAE performance. Future work will conduct statistically powered, cross-platform experiments with larger module sets, direct ITM/BAE instrumentation, contention workloads, UI-responsiveness testing, hierarchical composition, fault containment, policy verification, and adversarial security evaluation.

AI assistance disclosure. The author used OpenAI GPT-5.6 Thinking for language editing, consistency checking, and formatting recommendations.

REFERENCES

1. S. Salehi, I. Miremadi, M. Ghasempour Nejati, and H. Ghafouri, "Fostering the Adoption and Use of Super App Technology," IEEE Transactions on Engineering Management, vol. 71, pp. 4761–4775, 2024.
2. LY Corporation, "LINE MINI App API Reference," https://developers.line.biz/en/reference/line-mini-app/ (accessed July 17, 2026).
3. Tencent, "Weixin Mini Programs," https://www.tencent.com/products/weixin-mini-programs/ (accessed July 18, 2026).
4. Grab, "The Everyday Everything App," https://www.grab.com/my/ (accessed July 17, 2026).
5. M. Hasselwander and D. Weiss, "Key Factors Influencing Consumer Adoption Intentions of Super Apps in Germany," IEEE Access, vol. 12, 2024.
6. M. Yu, "The Rise of Super Apps: Insights and Opportunities from a UX Perspective," in 2023 IEEE International Professional Communication Conference, 2023.
7. L. Zhang, Z. Zhang, A. Liu, Y. Cao, X. Zhang, Y. Chen, Y. Zhang, G. Yang, and M. Yang, "Identity Confusion in WebView-Based Mobile App-in-App Ecosystems," in 31st USENIX Security Symposium, pp. 1597–1613, 2022.
8. Y. Zhang, Y. Yang, and Z. Lin, "Don't Leak Your Keys: Understanding, Measuring, and Exploiting the AppSecret Leaks in Mini-Programs," in Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, pp. 2411–2425, 2023.
9. S. Peltonen, L. Mezzalira, and D. Taibi, "Motivations, Benefits, and Issues for Adopting Micro-Frontends: A Multivocal Literature Review," Information and Software Technology, vol. 136, Art. no. 106571, 2021.
10. D. Taibi and L. Mezzalira, "Micro-Frontends: Principles, Implementations, and Pitfalls," ACM SIGSOFT Software Engineering Notes, vol. 47, no. 4, pp. 25–29, 2022.
11. Google, "Overview of Play Feature Delivery," Android Developers, https://developer.android.com/guide/playcore/feature-delivery (accessed July 17, 2026).
12. Apple, "Embedding Frameworks in an App," Technical Note TN2435, https://developer.apple.com/library/archive/technotes/tn2435/_index.html (accessed July 17, 2026).

13. FlutterFlow, “Scaling Super Apps: Modular Architecture with FlutterFlow Libraries,” July 27, 2025, https://flutterflow.io/blog/scaling-super-apps-modular-architecture-with-flutter-flow-libraries (accessed July 17, 2026).
14. Ionic, “What Is a Superapp?” https://ionic.io/resources/articles/superapps-composable-mobile-app-development (accessed July 17, 2026).
15. MobiWebX, “A SaaS IDE for Building Cross-Platform Applications,” https://aiot.com.tw (accessed July 17, 2026).
16. F. C. Cheng, E. Wang, A. Liu, A. Wu, P. C. Chung, Y. R. Pao, C. W. Wang, Z. H. Lin, J. Wang, and G. Y. Wang, “MobiWebX: A Mobile Applications and Web SaaS IDE with No-Code and Vibe-Coding Technologies,” ISBN 978-626-99759-0-7, MobiWebX Ltd., 2025.
17. A. H. Kusmanto and S. Surja, “Factors Influencing the Adoption of Digital Banking Service Super Apps in Indonesia Using Modified UTAUT2 Model,” in 2024 International Conference on Information Management and Technology (ICIMTech), pp. 695–700, 2024.
18. C. F. Zyvaristin, L. S. Saputra, and C. Hidayat, “Hooked on Convenience: The Loyalty Effect on Customers of E-Marketplace as Super Apps,” in 2025 8th International Conference of Computer and Informatics Engineering, pp. 1–6, 2025.
19. Andri, D. Y. K. Kristanto, E. N. Prastiwi, and A. E. J. Egeten, “User Experience Analysis of the BRImo Super Apps Using Usability Testing Method at PT. Bank Rakyat Indonesia,” in 2025 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS), 2025.
20. M. Nayebi and G. Ruhe, “Optimized Functionality for Super Mobile Apps,” in 2017 IEEE 25th International Requirements Engineering Conference, pp. 388–393, 2017.
21. B. Hu, Z. Liu, and Y. Chu, “Design and Implementation of Mini-App Shortcuts,” in 2024 4th International Conference on Electronic Information Engineering and Computer, pp. 195–206, 2024.
22. M. Yu, “Navigating Work–Life Communication Boundaries on Super Apps,” in 2024 IEEE International Professional Communication Conference (ProComm), pp. 153–159, 2024.
23. M. Christian, O. M. Muzammil, K. Gularso, S. Manurung, H. Yulita, and S. Wibowo, “Implementation of Logistics Super App: Impact on Technostress Creators, Employee Engagement, and Generational Roles,” in 2024 International Conference on Information Management and Technology (ICIMTech), pp. 145–157, 2024.



Fu-Chiung Cheng (鄭福炯) received the Ph.D. in Computer Science from Columbia University in 1998 and is an associate professor at Tatung University. His interests include no-code AI/IoT/mobile/web development, asynchronous circuits, and digital transformation.