ChatGPT

KMP SDK Research for Bring a Brain

Bringing AI-assisted, offline multiplayer dialogue to phones means designing a **distributed system** that spans iOS and Android. The research below covers BLE syncing, on-device AI interop, distributed state management, subscription validation and architectural patterns. Each section explains the relevant concepts and cites primary sources for the key technical facts.

1 Cross-Platform BLE protocol ("sync" engine)

1.1 MTU (Maximum Transmission Unit) optimisation

- **BLE payload limits:** The Bluetooth specification allows an attribute length of 512 bytes; however most stacks negotiate a smaller maximum transmission unit (MTU). Android 14 automatically requests an **ATT MTU of 517 bytes** when the first GATT client connects and ignores further requests [1]. In iOS the default packet size is **20 bytes** but recent versions allow negotiating up to **512 bytes** [2]. Using larger MTUs reduces overhead; tests show that increasing the MTU on Android can improve throughput by up to 15 % [3].
- **Chunking LLM output:** Because LLM responses can exceed the MTU, the sync engine should **chunk and reassemble** text. Treat each dialog line as a record with its own sequence number. On send, split the line into fragments smaller than the negotiated MTU minus protocol overhead; include fragment index and total count. On receive, reassemble fragments before dispatching to the state machine. When both devices support variable MTUs, use the smaller of the two.
- **Write type:** For time-sensitive data such as tokens from the LLM, use "Write Without Response" where possible. It yields higher throughput than acknowledged writes but is only available on characteristics without reliability requirements [2]. For voting or control packets where reliability matters, use Write With Response and await confirmation.

1.2 Peripheral vs central roles (host & peers)

- **Role definitions:** In BLE a **central** device scans for advertisements and initiates connections; a **peripheral** advertises and accepts connections. Smartphones typically act as centrals aggregating data from several peripherals [4]. However a device can implement both roles simultaneously.
- **Host responsibilities:** The Bring a Brain host will act as a **BLE peripheral**, advertising a *Bring a Brain Service*. Centrals (peers) initiate connections. To support up to three peers simultaneously, the host must **continue advertising while connected**; if advertising stops, only one peer may connect [5]. The host maintains an array of connected centrals and broadcasts scene changes or generated lines to each. After the session begins, peers may also send data to one another via the host.
- **Connection strategy:** When the host receives a join request, it assigns the peer an index (A,B,C). The host becomes the authoritative source of truth and ensures ordering of messages. When a peer becomes the host (e.g., original host disconnects), remaining devices elect a new host by comparing vector clocks (see §3). For networked play via WebSockets, the server acts as the host and Bluetooth roles are unused.

1.3 Service & characteristic mapping

Design a custom **GATT profile** for Bring a Brain following best practices from Novelbits on creating custom services [6] :

- **Service UUID:** Define a 128-bit UUID as the Bring a Brain service. Group related characteristics within this service. Reuse standard Generic Access/Attribute services for device name and appearance.
- **Command characteristic:** Writeable without response. Peers send control commands such as "Request Scene Change" or "Generate Next Lines." Each command includes the sender ID and Lamport timestamp (see §3). Use notifications to broadcast acceptance or rejection by the host.
- **Data characteristic:** Writeable without response. Carries chunks of dialog lines. Each fragment includes a header with `dialogId`, `speakerId`, `sequenceNumber`, `totalFragments` and vector clock. Receivers assemble fragments to reconstruct the line.
- **Vote characteristic:** Read/notify. Holds the current vote request and bitmask of responses. Peers update their bit in the bitmask via write with response. The host aggregates responses and notifies all peers of the result. Using a separate characteristic for votes reduces race conditions and allows concurrency with data transmissions.

Each characteristic should declare read/write/notify permissions appropriate for its function and use a short UUID derived from the base service UUID [7] .

1.4 Additional BLE considerations

- **Security & pairing:** For offline mode, use LE secure connections with Just Works or Numeric Comparison pairing. Exchange nonces during connection to prevent replay. Encrypt data packets using a symmetric session key derived from the pairing procedure.
- **Auto-reconnection:** Kable's reconnection logic should attempt to reconnect when devices move back in range. Devices maintain a short cache of pending operations to replay after reconnection. When the session times out, the host should clear the session state.

2 Local foundation model interop (Apple vs Android)

2.1 Kotlin `expect/actual` bridge to Apple's Foundation Models

- **On-device models:** Apple introduced a **3-billion-parameter language model** in iOS 26 that runs entirely on-device using 2-bit quantization [8] . Inference occurs on the Neural Engine; data never leaves the device [9] . The model is exposed through the **Foundation Models** framework. Developers check availability via `SystemLanguageModel.default` and create a `LanguageModelSession` for streaming responses [10] .
- **Expect/actual pattern:** In a KMP project, define an `expect class LocalAiEngine` in the shared module with functions like `fun prewarm()`, `fun generate(prompt: String): Flow<String>` and `fun dispose()`. On the iOS `actual` implementation, import the Swift class bridging into Kotlin via the `swiftklib` plugin. The bridging article shows how to create a Swift class and expose it to Kotlin using `@ObjCName` and cinterop; this pattern allows calling native frameworks from common code [11] . On Android, the `actual` implementation either calls a remote LLM via Ktor or uses a local Llama library compiled with Rust. Use `expect` to hide platform differences.

2.2 Prompt engineering portability & behaviour parity

- **System prompts:** Apple's on-device model uses generative guidelines similar to OpenAI models. However quantization and training data differences can affect tone and output. To maintain *behaviour parity* between offline (on-device) and online (Rust backend) modes:

- **Define a common system prompt** describing roles (e.g., AI Director), output format (native + translation), and persona guidelines.

- **Calibrate responses** by testing the prompt on both Apple's model and the server LLM. Adjust phrasing, temperature, and maximum tokens until the style and content align. For example, emphasise brevity or use explicit translation separators.

- **Implement fallback heuristics**: if the local model fails or returns unexpected output (e.g., due to limited context window of 4096 tokens [12]), request generation from the remote server.

- **Streaming vs. batch:** `LanguageModelSession` supports streaming tokens; the server API should mimic this by sending partial tokens over WebSocket so the UI can display progressive typing. When chunking for BLE, group tokens into ~200-byte packets to avoid fragmentation.

2.3 Resource management & memory pressure

Running an on-device LLM while maintaining multiple BLE connections can exhaust memory. Research indicates:

- **Memory usage:** The 3B model uses 2-bit quantization; memory consumption is reduced by **~37.5 %** compared with 8-bit [8], but it still requires hundreds of megabytes. On iOS, the Foundation Models framework provides a `prewarm()` method to load resources ahead of time and reduce latency [12].
- **Kotlin/Native GC:** The Kotlin/Native runtime uses a concurrent garbage collector that reacts to **memory pressure heuristics**. You can manually trigger the GC via `kotlin.native.internal.GC.collect()` and monitor GC pauses with Xcode Instruments [13]. Developers can enable memory tagging to track allocations and use the VM Tracker to view Kotlin memory usage [14].
- **Monitoring strategy:** Instrument the AI provider to report memory usage after generating responses. When memory reaches a threshold, throttle BLE transfers and prefetch fewer lines. On iOS, `LanguageModelSession` provides progress events; cancel early if memory becomes constrained.

3 Distributed state synchronization (MVI + conflict resolution)

In multiplayer offline mode, multiple devices may attempt to generate or change scenes concurrently. To avoid forks, the app must establish a logical ordering of events and only apply state changes after consensus.

3.1 Logical clocks: Lamport vs vector clocks

- **Lamport clocks:** Each device maintains a counter. When an internal event (user action) occurs or a message is sent, increment the counter. Each message includes the sender's clock value. Upon receiving a message, update the local clock to `max(received, local) + 1` [15]. Lamport clocks provide a total ordering but cannot detect concurrency.

- **Vector clocks:** Each device maintains an array with an entry per device. On a local event, increment its own index. When sending a message, include the entire vector. On receive, update each element to the **element-wise maximum** of the local and received vectors and then increment the receiver's own index [16]. Comparing vectors allows determining if one event happened before another or if they are concurrent [17]. The trade-off is increased message size (O(n) for n devices).

Recommended approach: Use **vector clocks** because up to four devices have manageable vectors (size 4). Each dialog event (Generate, Vote, Scene change) carries the sender's vector clock. On the host, insert events into a queue ordered by vector clock; break ties by device index. When two events are concurrent (no causal relationship), apply a deterministic conflict resolution (e.g., lexical order of device IDs) or require a vote.

3.2 Network-aware MVI reducer

MVIKotlin provides a unidirectional data flow but does not handle network replication [18]. To make the store network-aware:

- **Integrate logical clocks:** Extend `SessionState` to include the current vector clock. When an `Intent` is received from the UI, wrap it in a `PendingAction` with the current clock. The host replicates this action to peers via BLE/WebSocket; peers buffer actions until receiving them from the host, then apply them in order. State changes occur only after the host acknowledges the action.
- **Consensus mechanism:** For operations requiring unanimity (e.g., scene changes), host sends a `VoteRequest` and waits for votes. When all peers vote 'Yes,' host updates the state and notifies peers through the vote characteristic. If a peer votes 'No' or times out, the host rejects the request.
- **Conflict resolution:** When two actions are concurrent and cannot be ordered by vector clocks, choose the action with the smaller device ID; the loser discards its local result and adopts the winner's state. This deterministic strategy ensures eventual consistency.

3.3 Synchronized navigation

The UI navigation is handled by **Decompose**, a KMP library for lifecycle-aware components and cross-platform navigation [19]. To ensure all players see the same screen:

- Synchronize the **navigation state** (`DialogPhase` or screen key) through the BLE data channel. When the host changes phase (e.g., from Onboarding to Scenario selection), broadcast the new phase along with a vector clock. The `MVI` reducer updates `currentPhase` only after the message is confirmed.
- Use Decompose's back-stack mechanism to construct navigation controllers in shared code. The UI layers on iOS/Android observe the state flow; therefore, a single change in `SessionState.currentPhase` is enough to navigate on all devices.

4 Subscription & receipt validation

4.1 StoreKit 2 & JWS extraction

- **JWS in StoreKit 2:** When a purchase or renewal occurs, the `Transaction` object includes a **JWS representation**. Apple's `verificationResult.jwsRepresentation` property returns the signed transaction, which should be sent to the backend for validation [20]. When monitoring

subscription status, update the stored JWS whenever the `SubscriptionInfo.Status` transitions to `subscribed` or `inGracePeriod` [21].

- **Server verification:** On the backend, use Apple's root certificates to verify and decode the JWS. Then inspect fields like `expiresDate` and `revocationDate` to determine whether the subscription is active [22]. This ensures the server can trust the client's receipt.

4.2 Android billing & purchase tokens

- **Acknowledge purchases:** With Google Play Billing Library 7.0+, after a purchase is made, call `acknowledgePurchase()` with an **AcknowledgePurchaseParams** object containing the `purchaseToken` [23]. For consumables, call `consumeAsync()` with a `ConsumeParams` built from the same token [24].
- **Verify on server:** Send the `purchase.purchaseToken` to your backend. After verifying with Google's API, acknowledge or consume the purchase. The verification must occur within three days to avoid automatic refunds [25].

4.3 Grace period & account linking

Google offers a **grace period** when auto-renewal fails. The subscription enters `SUBSCRIPTION_STATE_IN_GRACE_PERIOD`; developers can configure durations between 3 – 30 days [26]. During this period, the user retains access; use In-App Messaging to prompt for updated payment methods [27]. Even with a zero-day grace period, Google provides a one-day silent grace period [28]. If the user still fails to pay, the subscription transitions to `ACCOUNT_HOLD`, which can last up to 60 days minus the grace period [29].

Account linking: When a user purchases on iOS but wants to use an Android tablet, store the subscription status in your Rust backend keyed by the user's account. On Android login, query the backend; if a valid receipt exists, treat the user as premium and skip local purchase. On iOS, provide a restore purchases flow that fetches the latest transaction and revalidates the JWS.

5 Architectural patterns

5.1 Repository pattern & data-source switching

The repository layer acts as a **single source of truth** and abstracts multiple data sources. Android's architectural guide states that each repository should **expose data**, **centralize changes**, **resolve conflicts** and **abstract data sources** [30]. A repository should depend on data sources for remote (network), local (database) or other sources and use dependency injection to pass them in [31]. The rest of the app (UI or domain layers) should never interact with data sources directly [32].

For Bring a Brain, implement a `DialogRepository` interface in shared code and create these implementations:

Data source	Description	When to use
LocalAiDataSource	Wraps on-device Foundation Model (iOS) or offline Llama library (Android). Provides `generate(dialogState)` and caches previous dialogs in `SQLDelight`.	Offline + free play.

Data source	Description	When to use
BleDataSource	Handles BLE transmission. Provides functions like `broadcastAction()`, `sendChunks()`, `requestVote()` and streams remote actions.	Multiplayer offline mode.
RemoteApiDataSource	Calls the Rust backend via Ktor for text generation, history sync and subscription verification.	Online mode or when local model fails.

`DialogRepositoryImpl` decides which data source to use based on `SessionState.connectionStatus` and `isPremium`. It listens to flows from all sources and merges them into a single `Flow<DialogEvent>`. When network conditions change (e.g., online becomes offline), it switches to the appropriate source and updates the store accordingly.

5.2 Serialization format for BLE (Protobuf vs JSON)

- **Binary vs text:** JSON is human-readable but uses a text-based encoding, leading to larger payloads and slower serialization for complex structures [33]. Protocol Buffers (Protobuf) use a **binary encoding** and enforce a strict schema. Benefits include **compact data size**—smaller messages reduce bandwidth and latency—and **high performance** in serialization and deserialization [34]. Protobuf also supports schema evolution via explicit field numbers [35].
- **BLE considerations:** Because BLE packets are limited to a few hundred bytes, Protobuf's compact encoding reduces fragmentation. Schema enforcement catches version mismatches early and adds a layer of security [36]. JSON may be suitable for debugging and initial development but should be avoided for on-air transmissions.
- **Implementation:** Use Kotlin Serialization with the Protobuf encoder to serialize message envelopes (command, data, vote) into byte arrays. Define `.proto` files in `commonMain` and generate Kotlin/Swift/Rust classes. For debugging, provide a JSON adaptor that wraps the same data classes.

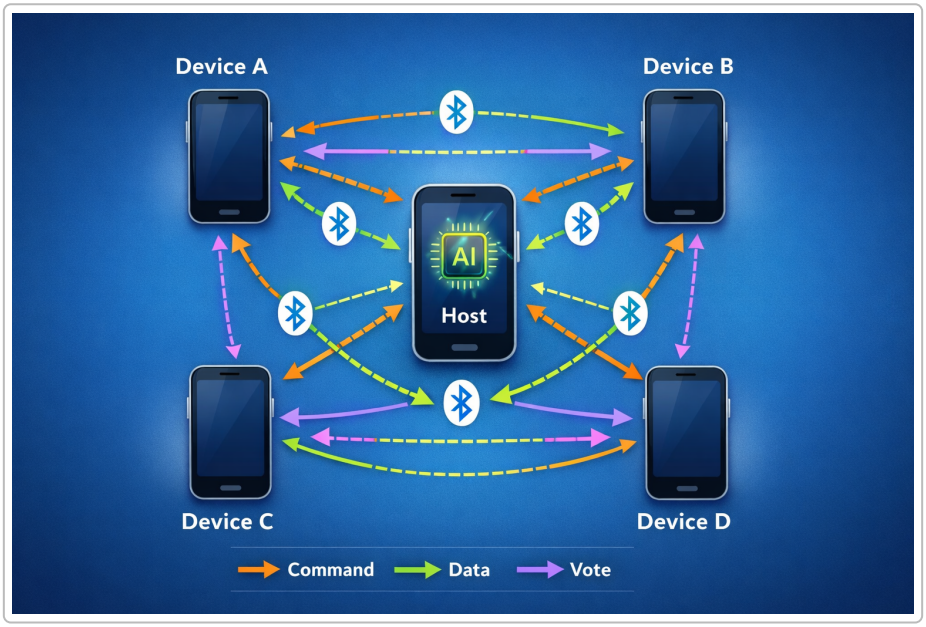
Summary table: research priorities & tasks

Priority	Topic & tasks	Key tool
Highest	**BLE state sync** – negotiate MTU; implement chunking/assembly; design GATT profile with command, data and vote characteristics; implement host advertising and multi-connection support; secure pairing; handle reconnections.	Kable + conflict-resolution logic
High	**On-device AI bridge** – create `expect/actual` `LocalAiEngine` bridging to Swift `LanguageModelSession` and Android remote API; calibrate prompts for behaviour parity; implement streaming generation; prewarm models; monitor memory pressure and throttle BLE transfers.	Foundation Models & Kotlin/Native GC
Medium	**Distributed state sync** – use vector clocks to order events; extend MVIKotlin reducer to apply actions only after host consensus; implement voting mechanism with majority/all-approve semantics; integrate Decompose navigation sync.	MVIKotlin + vector clocks

Priority	Topic & tasks	Key tool
Medium	**Unified auth/billing** – extract JWS receipts in StoreKit 2; send to Rust backend for verification; handle Google Play purchase tokens; implement restore/acknowledge flows; manage grace periods and account holds; link subscriptions across devices via backend.	StoreKit 2 + Google Billing Library
Low	**Repository & serialization** – design `DialogRepository` with pluggable data sources; use Protobuf for BLE transmissions; fallback to JSON only for debugging; ensure repository acts as single source of truth.	Repository pattern + Kotlin Serialization

Conceptual diagram

The following illustration visualises the peer-to-peer sync architecture. Four devices connect via Bluetooth; the central host (middle) manages commands (orange arrows), data (green arrows) and votes (purple arrows).



Conclusion

Building Bring a Brain as a KMP SDK requires combining mobile BLE networking, on-device AI, distributed systems theory and subscription infrastructure. The research above highlights the technical constraints (MTU limits, quantized models, logical clock algorithms, billing flows) and proposes concrete tasks and patterns to guide implementation.

---

1 BluetoothGatt.RequestMtu(Int32) Method (Android.Bluetooth) | Microsoft Learn
https://learn.microsoft.com/en-us/dotnet/api/android.bluetooth.bluetoothgatt.requestmtu

2 ⚡ Optimizing BLE Performance in iOS: Best Practices for Speed, Battery Life, and Reliability
https://www.linkedin.com/pulse/optimizing-ble-performance-ios-best-practices-speed-battery-vipin-etp3e

3 Maximizing BLE Throughput Part 4: Everything You Need To Know – Punch Through
https://punchthrough.com/ble-throughput-part-4/

4 Bluetooth Low Energy: Central vs. Peripheral | IoT For All
https://www.iotforall.com/bluetooth-low-energy-central-vs-peripheral

5 The Ultimate Guide To Managing Your BLE Connection – Punch Through
https://punchthrough.com/manage-ble-connection/

6 7 Bluetooth GATT: How to Design Custom Services & Characteristics
https://novelbits.io/bluetooth-gatt-services-characteristics/

8 9 10 Apple Just Handed Every Developer a 3-Billion Parameter AI Model, No Cloud Required | by Cogni Down Under | Medium
https://medium.com/@cognidownunder/apple-just-handed-every-developer-a-3-billion-parameter-ai-model-no-cloud-required-b2118eaac574

11 How to Use Swift Inside Kotlin Multiplatform: The iOS Bridge Explained (with App Review Dialog as Example) | by Houssam Eddine Baba Bendermel | Medium
https://medium.com/@Tweeel/how-to-use-swift-inside-kotlin-multiplatform-the-ios-bridge-explained-with-a-real-example-63fea919d355

12 The Ultimate Guide To The Foundation Models Framework | AzamSharp
https://azamsharp.com/2025/06/18/the-ultimate-guide-to-the-foundation-models-framework.html

13 14 Kotlin/Native memory management | Kotlin Documentation
https://kotlinlang.org/docs/native-memory-manager.html

15 16 17 Logical Clocks - Documentation
https://aeron.io/docs/distributed-systems-basics/logical-clocks/

18 MVIKotlin
https://arkivanov.github.io/MVIKotlin/

19 Overview - Decompose
https://arkivanov.github.io/Decompose/

20 21 22 How to Validate iOS and macOS In-App Purchases Using StoreKit 2 and Server-Side Swift | by Ronald Mannak | Medium
https://medium.com/@ronaldmannak/how-to-validate-ios-and-macos-in-app-purchases-using-storekit-2-and-server-side-swift-98626641d3ea

23 24 25 Processing in-app purchases with Google Play Billing Library
https://adapty.io/blog/android-in-app-purchases-with-google-play-billing-library/

26 27 28 29 Understanding Google Play's subscription lifecycle: a complete guide
https://www.revenuecat.com/blog/engineering/google-play-lifecycle/

30 31 32 Data layer | App architecture | Android Developers
https://developer.android.com/topic/architecture/data-layer

33 34 35 36 Protobuf vs JSON: Performance, Efficiency & API Speed
https://www.gravitee.io/blog/protobuf-vs-json