ChatGPT

TripCactus: AI-powered trip planner PRD

1. Purpose & background

The **Mobile AI Hackathon** hosted by Cactus, Nothing and Hugging Face invites Flutter, React Native and Kotlin developers to build local, private and personal AI apps [1]. The hackathon emphasises that cloud AI is powerful but **"the future belongs to the Edge"** [2]. Submissions must use the **Cactus SDK** and demonstrate on-device AI that provides **total privacy**, **zero latency** and **offline capability** [3]. The event offers a **Main Track** for the best mobile AI app, a **Memory Master** track for shared on-device knowledge bases, and a **Hybrid Hero** track for local/cloud routing strategies [4]. Successful entries must deliver a working build and show local AI functionality [5].

TripCactus is a local-first trip companion aligned with these goals. It helps travellers plan, organise and query their trips using on-device language models. The app enables itinerary generation (via a privacy-preserving cloud planner), offline timeline adjustments and Q&A using a shared trip memory. It targets the **Main Track** by building a real mobile app with the Cactus SDK. It addresses **Track 1** by implementing a local knowledge base with memory chunks and a logistics graph. It covers **Track 2** by routing certain tasks (global itinerary generation) to the cloud while keeping personal data local.

2. Success criteria & hackathon alignment

To maximise scoring, TripCactus is designed around the hackathon's implied evaluation categories. These categories are interpreted from the organiser's requirements and general hackathon practice:

- **Technical Implementation** – Use the React Native Cactus SDK correctly. Demonstrate model download, embedding and completion using on-device models. Use the Cactus hook (`useCactusLM`) for completions and embeddings [6]. Showcase hybrid mode fallback where appropriate [7].
- **Edge Capabilities** – Ensure that critical features run offline. The app must show total privacy (no personal data leaves the device), zero-latency responses and offline functionality [3].
- **Design & UX** – Provide a polished timeline view, intuitive trip creation and chat interface. Communicate model download status in the UI using Cactus hooks (e.g., `isDownloading` and `downloadProgress`) [6].
- **Utility & Innovation** – Solve real travel problems such as itinerary planning, local Q&A and schedule adjustments without internet. Implement a novel logistics graph and local memory store for reasoning about time and location.
- **Completeness** – Deliver a fully functioning app with at least two hero features and an accompanying `.apk`/`.ipa`. Provide a working cloud planner endpoint (can be a simple HTTP function) and local memory persistence.

3. User flows & UI views

TripCactus follows the flows defined by the team:

1. **Initial load (Screen 1)** – List existing trips stored locally. Provide a **“Start new trip”** button. Show a spinner if the Cactus model is downloading.
2. **Trip view (Screen 2)** – When the user enters a trip (new or existing), display a **blank timeline** showing days between the trip’s start and end. Include a **settings button** to edit trip metadata (name, dates, home airport) and to add manual items (flights, lodging, activities). Include a **“Generate itinerary with AI”** button that triggers the hybrid day planner.
3. **Chat view (Screen 3)** – Accessed from a chat icon on the trip view. Shows a chat history with the “Trip Brain.” Allows the user to ask questions about the itinerary. When offline, answers come from on-device context; when online, context is updated after cloud calls. The chat can also accept commands to modify the schedule (e.g., “Move dinner to the morning”).

These flows map directly to the features below.

4. Feature descriptions

4.1 Hero Feature 1 – Offline Trip Brain (Memory Master)

Value proposition: Provide travellers with an offline “trip brain” that can answer questions and adjust the schedule based on a local knowledge base. The feature supports Track 1 by implementing shared memory and retrieval-augmented generation on-device.

Key capabilities:

- **Structured itinerary storage** – Trips consist of `Trip`, `DayPlan` and `TripItem` entities stored locally via SQLite or AsyncStorage. Items include flights, lodging, activities and transport, each with start/end times, location tags and optional metadata.
- **Memory chunks** – For each `TripItem` (or day summary) a canonical sentence is generated (e.g., “Flight from Toronto (YYZ) to Tokyo (HND) on 2025-12-01 departing 10:35 and arriving 15:20”) and embedded using the local model’s `embed` method [6]. The resulting vector and text are stored in `MemoryChunk` records.
- **Logistics graph** – Derived from trip items and place metadata. Nodes include days, events and places; edges encode temporal order (`BEFORE`/`AFTER`), membership (`WITHIN_DAY`), anchors (flights and hotels), and spatial relations (`NEAR`, `SAME_AREA`). A `Place` table stores simple geospatial information (city, area tags, `near` list) rather than full maps.
- **Local Q&A** – When the user asks a question, the system embeds the question using `embed` [6], retrieves the top `k` memory chunks by cosine similarity, and computes additional context from the logistics graph (e.g., next events after a given time). The app constructs a prompt instructing the local model to answer solely from the provided context. If the answer is not in the context, the model is told to admit it does not know.
- **Offline modifications** – Users can issue commands like “Move my sushi dinner to noon.” The system identifies the relevant `TripItem`, adjusts its start/end times while preserving ordering, updates the logistics graph, re-embeds the canonical sentences and persists the changes.

Success measurement: The feature should work without network connectivity after the model has been downloaded. All itinerary data and embeddings remain on device, ensuring privacy and low latency. Chat responses should reflect the updated schedule immediately. Answers outside the stored context should prompt the user to add missing information rather than hallucinate.

4.2 Hero Feature 2 – Hybrid Day Planner (Hybrid Hero)

Value proposition: Generate daily itineraries that leverage real-world data when online but still operate offline once downloaded. This demonstrates a router pattern for Track 2.

Key capabilities:

- **User input & preferences** – On the trip view, the user enters a city, date, available time ranges and optional interests (e.g., food, history). This minimal dataset does not include personal identifiers.
- **Router logic** – A simple router determines whether to call the cloud planner:
- If generating a plan from scratch or regenerating with new preferences **and** the device has internet, it sends a cloud request. Only coarse data (city, date, time ranges, preferences) is transmitted.
- For minor edits or offline use, it falls back to a local re-planner that operates on existing trip items.
- **Cloud planner** – A small external service (e.g., Cloudflare Worker) uses a large language model to generate a JSON itinerary (title, start time, end time, place tags). No user identifiers or embeddings are sent. The response is parsed, inserted as `TripItem`s and converted into memory chunks for offline use.
- **Local re-planning** – Users can ask the local model to adjust an existing plan (e.g., "Make my afternoon free"). The system builds a context summarising the day's events and sends it to the local model with instructions to rearrange within existing activities. After receiving the new order, the DB and memory store are updated accordingly.
- **Hybrid fallback** – When running completions, the app may use Cactus' **hybrid mode** (`mode: 'hybrid'`) to automatically fall back to a cloud provider if local inference fails [7]. This demonstrates the hybrid strategy recommended by the hackathon.

Success measurement: The planner must generate a reasonable day itinerary when online and gracefully handle offline conditions. Plans should integrate seamlessly with the local memory store so that subsequent queries about the day work offline. The network payload must exclude any personal data, highlighting the privacy story.

4.3 Optional Feature – Translation & Notes

If time permits, implement a lightweight translation tool using the local model or built-in Cactus STT/ embedding features [8]. Users can translate phrases or record notes, which are stored as additional memory chunks and can be recalled by the Trip Brain.

5. Data models & boundaries

5.1 Structured data

The following TypeScript-style types define the core data stored locally. All personal data remains on device.

```
type Trip = {
  id: string;
  name: string;
  startDate: string; // ISO date e.g. '2025-12-01'
  endDate: string;
  homeAirport?: string;
};

type DayPlan = {
  id: string;
  tripId: string;
  date: string; // YYYY-MM-DD
};

type TripItemType = 'FLIGHT' | 'LODGING' | 'ACTIVITY' | 'TRANSPORT' | 'NOTE';

type TripItem = {
  id: string;
  tripId: string;
  dayId?: string;
  type: TripItemType;
  title: string;
  placeId?: string;
  city?: string;
  areaTag?: string;
  startDateTime?: string; // ISO timestamp
  endDateTime?: string;
  meta?: {
    airline?: string;
    flightNumber?: string;
    confirmationCode?: string;
    hotelName?: string;
    address?: string;
    notes?: string;
  };
};

type Place = {
  id: string;
  name: string;
  city: string;
  areaTags: string[];
  near: string[]; // list of Place ids or area tags
};

type MemoryChunk = {
  id: string;
  tripId: string;
  itemId?: string; // link to TripItem
  kind: 'ENTRY' | 'DAY_SUMMARY' | 'NOTE';
  text: string;
  embedding: number[];
};
```

5.2 Derived structures

- **Logistics graph** – Derived on-the-fly from `TripItem` and `Place`. Edges are not persisted but computed when answering queries:
- `WITHIN_DAY`: links a `DayPlan` to its items.
- `BEFORE` / `AFTER`: chronological order of items on a day.
- `ANCHOR`: marks flights and lodging.
- `AT_PLACE`: ties a `TripItem` to a `Place` via `placeId`.
- `NEAR`: uses the `near` field of `Place` to suggest nearby points without maps.
- `SAME_AREA`: identifies items in the same city or area tag.

- **Memory store** – A component that indexes `MemoryChunk` embeddings and supports similarity search. It may use a naïve cosine similarity algorithm or a small ANN index. It must remain entirely on device.

5.3 Context boundaries

- **On-device only** – All personal trip data (`TripItem` fields, embeddings, memory chunks, chat history) stays on device. Local inference uses only these data. No raw itinerary data or personal identifiers are sent to the cloud.
- **Cloud-visible data** – Only non-identifying parameters required for planning (city names, dates, coarse time ranges, high-level interests) are sent to the cloud service.
- **Model context** – When constructing prompts, only the retrieved memory chunks and the specific question are passed to the model. The system instructs the model not to use any other knowledge and to admit uncertainty if necessary.

6. System prompt design for Trip Brain

The local language model needs clear instructions to avoid hallucinations. An example system prompt (pseudo-content) is shown below. It should be prepended to every chat call when answering or modifying itineraries. The prompt uses the memory context and optionally logistic information.

```
SYSTEM:
You are **TripBrain**, a local travel assistant.  You can only answer using the
trip context provided below.  If the answer is not in the context, reply “I
don’t know, please add it.”  Follow these rules:

* Use 24-hour time and the trip’s local timezone.
* Do not fabricate any details that are not in the context.
* If the user asks to move or edit an event, describe the updated timeline and
ask for confirmation before saving.
* When listing events, include date, time and title.

User question and context will follow.

USER:
Trip context:
<insert top-k memory chunks and derived logistic notes>

Question: <insert the user’s question here>
```

For modification commands, the system prompt should instruct the model to return a new ordered list of events with updated times only using existing activities. The controller code will then parse the response and update the DB accordingly.

7. Pseudocode pipelines for key queries

Below are high-level pseudocode pipelines for common questions. Each pipeline is broken into deterministic steps (data retrieval and graph reasoning) and an LLM call.

7.1 Query: “What can I do after [Place/Event]?”

```
function answerWhatAfter(question, tripId):
  // Step 1: Identify referenced event or place
  qEmbedding = embed(question)  // local embed [6]
  candidates = memoryStore.search(tripId, qEmbedding, topK=5)
  targetItem = findFlightOrActivityMentioned(candidates, question)

  if not targetItem:
    return "I don’t know, please specify the event."

  // Step 2: Gather subsequent events
  eventsAfter = getItemsAfter(tripId, targetItem.dayId,
targetItem.startDateTime)

  // Step 3: Gather nearby suggestions from logistics graph
  place = getPlace(targetItem.placeId)
  nearbyPlaceIds = place.near + findPlacesWithSameArea(place.areaTags)
  suggestedItems = filter eventsAfter where item.placeId in nearbyPlaceIds

  // Step 4: Build context and call model
  contextChunks = buildContextChunks(eventsAfter, suggestedItems)
  prompt = buildSystemPrompt(question, contextChunks)
  response = completeLocal(prompt)  // local completion
  return response
```

7.2 Query: “When is my flight to [Destination]?”

```
function answerFlightTime(question, tripId):
  // Step 1: Semantic search for flight items
  qEmbedding = embed(question)
  candidates = memoryStore.search(tripId, qEmbedding, topK=5)
  flight = findFlightToDestination(candidates, question)
  if not flight:
    return "I don’t know, please add your flight details."

  // Step 2: Build a simple structured answer without LLM
  // to guarantee determinism
  departure = formatDateTime(flight.startDateTime)
  arrival = formatDateTime(flight.endDateTime)
  return f"Your flight to {flight.city} departs at {departure} and arrives at
{arrival}."
```

7.3 Command: “Move [Activity] to the morning”

```
function handleMoveActivity(question, tripId):
  // Step 1: Semantic search to identify the activity
  qEmbedding = embed(question)
  candidates = memoryStore.search(tripId, qEmbedding, topK=5)
  activity = findActivityMentioned(candidates, question)
  if not activity:
    return "I don’t know which activity to move."

  // Step 2: Compute new time slot
  dayId = activity.dayId
  morningSlot = findEarliestAvailableSlot(tripId, dayId, duration(activity))
  if not morningSlot:
    return "No free morning slot available."

  // Step 3: Update the DB and memory
  activity.startDateTime = morningSlot.start
  activity.endDateTime = morningSlot.end
  upsertTripItem(activity)  // updates DB and memory chunks

  // Step 4: Build context and describe new schedule using LLM
  updatedEvents = getDayItems(dayId) sorted by startDateTime
  context = summariseDay(updatedEvents)
  prompt = buildReplanPrompt(context, question)
  response = completeLocal(prompt)
  return response
```

7.4 Query: "Do I have free time between 14:00 and 17:00 near my hotel?"

```
function checkFreeTimeNearHotel(question, tripId):
  // Parse time range and day from question (simple regex)
  requestedStart, requestedEnd, day = parseTimeRange(question)
  // Identify hotel for that day
  hotel = findHotelForDay(tripId, day)
  // Compute free blocks
  occupied = getIntervalsForDay(tripId, day)
  freeBlocks = invertIntervals(occupied)
  // Check if requested interval fits within a free block
  fits = any(block.start <= requestedStart and block.end >= requestedEnd for
block in freeBlocks)
  // Suggest nearby places if free
  if fits:
    place = getPlace(hotel.placeId)
    suggestions = findItemsNear(place.areaTags, day, freeBlocks)
    context = buildContext(freeBlocks, suggestions)
    prompt = buildSystemPrompt(question, context)
    return completeLocal(prompt)
  else:
    return "You do not have a free window at that time."
```

8. Implementation plan & milestones

Given the hackathon's time constraints (~24 hours), implementation should be broken into focused phases. Each phase can be assigned to a separate coding agent.

Phase 0 – Project setup (1 hour)

1. Initialise an Expo React Native project with bare workflow to use native modules.
2. Install `cactus-react-native` and `react-native-nitro-modules` [9]. Confirm the model downloads properly using the quick start example [10].
3. Set up SQLite or AsyncStorage for persistent data.

Phase 1 – Data & memory layer (3 hours)

1. Implement the TypeScript models (`Trip`, `DayPlan`, `TripItem`, `Place`, `MemoryChunk`).
2. Create CRUD utilities for storing and retrieving these from the local database.
3. Implement `canonicalizeItem(item: TripItem): string` that generates a descriptive sentence for each item.
4. Build the memory store: index embeddings and provide a `search(tripId, embedding, topK)` method.

5. Seed the database with a sample trip for testing.

Phase 2 – Logistics graph & helper functions (3 hours)

1. Write functions to derive ordering relationships and find available time slots. For example, `getItemsAfter(dayId, time)` and `findEarliestAvailableSlot()`.
2. Implement basic `Place` lookup and `near` suggestions. Pre-load a small set of sample places for demonstration.
3. Write time parsing helpers to extract ranges from user questions.

Phase 3 – Trip Brain UI & chat integration (4 hours)

1. Implement Screen 1: list trips and start new trip.
2. Implement Screen 2: timeline view with settings and "Generate itinerary" button. Show model download progress using `isDownloading` and `downloadProgress` properties [6].
3. Implement Screen 3: chat interface. Use Cactus hooks (`useCactusLM`) for local completions. Wire the memory search and logistics graph to build context and prompts. Display streaming responses if time allows.
4. Implement offline modifications: update DB and memory when the user issues a command.

Phase 4 – Hybrid planner (5 hours)

1. Implement a simple HTTP function that accepts (city, date, time ranges, interests) and returns a JSON itinerary. Use any cloud LLM API (e.g., OpenAI or Hugging Face) to generate the plan and structure it. Ensure it does not send any user PII.
2. In the app, implement router logic to decide whether to call the planner or run local re-planning. Use Cactus hybrid mode (`mode: 'hybrid'`) for fallback [7].
3. Parse the planner response into `TripItem`s and insert them into the DB. Call `embed()` on each sentence to update the memory store.
4. Implement local re-planner logic using the pseudo-pipeline from §7.3.

Phase 5 – Polish & video demo (time permitting)

1. Add optional translation tool using `CactusSTT` for speech-to-text [8] and local completions to translate phrases.
2. Add icons, animations and clear state indicators (e.g., offline badge).
3. Record a one-minute demo video: show model download, generate an itinerary, go offline, ask Trip Brain questions, adjust schedule, and highlight privacy/offline features.

9. Non-goals and constraints

To ensure scope discipline, the following are explicitly out of scope:

- Full-fledged mapping or navigation. The app does not compute travel times or directions.
- Live flight/hotel bookings or API integrations. All data entry is manual or via user input.
- Comprehensive offline POI databases. Only a small curated list is shipped for demonstration.
- Multi-user synchronisation. The hackathon prototype supports only single-device use.
- Security hardening or encryption beyond OS storage. It can be added post-hackathon.

10. Future opportunities

After the hackathon, TripCactus could evolve with:

- A richer offline POI database and optional map view.
- Agentic tool-calling via Cactus to automate bookings or weather checks.
- Multi-day summarisation and automatic journalling features.
- Integration with third-party calendars for import/export.

11. References

- The hackathon's mission emphasises building mobile apps that leverage on-device AI and achieve total privacy, zero latency and offline capability [3].
- Participants must use the **Cactus SDK** and build with frameworks including React Native, Flutter and Kotlin [11].
- The hackathon's themes include **Memory Master** (shared knowledge base) and **Hybrid Hero** (local/cloud routing) [4].
- Example code for installing and using `cactus-react-native` shows downloading models and generating completions [10].
- The Cactus SDK supports text embedding via `embed()` and provides hooks for React Native [6].
- Hybrid mode allows completions that fall back to a cloud provider if local inference fails [7].
- Cactus also provides speech-to-text functions such as `transcribe()` using Whisper models [8].

---

[1] [2] [3] [4] [5] [11] Mobile AI Hackathon: Cactus X Nothing X Hugging Face · Luma
https://luma.com/jrec73nt

[6] [7] [8] [9] [10] React Native SDK
https://cactuscompute.com/docs/react-native