Building an Offline-First Emergency Preparedness App: Complete Technical Specifications

1. Core Architecture and User Experience

1.1. Three-Layer Interaction Model

The application's user experience is built upon a three-layer architecture designed to provide intelligent, context-aware assistance while maintaining a fully offline-first posture. This model prioritizes speed, accuracy, and reliability, ensuring that users can access the information they need in a crisis without relying on network connectivity.

1.1.1. Layer 1: On-Device Natural Language Intent Classification

The first layer of interaction is a lightweight, on-device machine learning model responsible for understanding the user's intent. When a user types a query in natural language, such as "I'm cold and wet" or "How do I treat a burn?", this layer analyzes the input and classifies it into a predefined category of emergency. The primary technology for this is **Apple's NaturalLanguage framework with MLTextClassifier**, which offers a powerful yet compact solution. This approach avoids the massive size and computational overhead of large language models (LLMs), making it ideal for a mobile application with a strict 500MB budget. The classifier is trained on a dataset of common emergency queries and is designed to be fast and accurate, with an inference latency of just **3-10ms** on the A18 Pro's Neural Engine. This ensures that the app can respond to user input almost instantaneously, a critical feature in a time-sensitive emergency.

1.1.2. Layer 2: Pre-Authored Conversation Trees for Critical Scenarios

Once the user's intent has been classified, the second layer of the architecture takes over. For high-stakes, time-critical scenarios like cardiac arrest, severe bleeding, or choking, the app transitions from a simple information retrieval tool to an interactive guide. This is achieved through **pre-authored conversation trees** written in the **Ink scripting language**. These are not static documents but dynamic, branching narratives that guide the user through a series of questions and actions. The Ink format allows for state tracking, meaning the app can "remember" previous answers and tailor subsequent instructions accordingly. For example, in a medical triage scenario, the app might ask about the victim's consciousness, and based on the answer, it will branch to a different set of instructions. This conversational approach provides a more supportive

and less overwhelming experience than a simple list of instructions, helping to calm the user and ensure that critical steps are not missed.

1.1.3. Layer 3: Offline Search and Online API Fallback

The third layer of the architecture provides access to a vast repository of reference information for less time-critical queries or for situations where the user needs more detailed information. This layer is powered by a comprehensive offline database built with **SQLite and the FTS5 full-text search extension**. This allows users to search through thousands of documents—including medical guides, legal rights, and survival manuals—with sub-10ms latency. To enhance the search experience beyond simple keyword matching, the database also incorporates **semantic search** using pre-computed vector embeddings, enabling the app to find relevant information even if the user's query does not contain the exact keywords. While the app is designed to be fully functional offline, this third layer also includes an **online API fallback**. When a network connection is available, the app can query online sources for the most up-to-date information, such as real-time weather alerts or emergency service updates, providing an enhanced layer of functionality without compromising the core offline experience.

1.2. Conversational Interface Design

1.2.1. Utilizing the Ink Scripting Language for Branching Narratives

The selection of a scripting language for authoring interactive, branching narratives is a critical architectural decision for the emergency preparedness app. After evaluating several options, **Ink**, developed by Inkle Studios, emerges as the most mature and suitable choice for this use case . Ink is a purpose-built, open-source scripting language designed specifically for creating interactive, branching text-based narratives and dialogue trees, making it ideal for guiding users through complex emergency scenarios like medical triage. Its proven track record is demonstrated by its use in award-winning, narrative-rich games such as "80 Days" and "Highland Song," which attests to its power and flexibility in handling dynamic storytelling . The language's design prioritizes a clean, text-based markup syntax that is both human-readable and highly expressive, allowing content creators to define intricate story flows, conditional logic, and state tracking without deep programming expertise. This is a significant advantage for an emergency app, where the accuracy and clarity of the guidance are paramount. The ability to manage variables and conditional logic within the script enables the creation of personalized, context-aware conversations that adapt based on user inputs, a crucial feature for effective emergency response protocols.

A key technical advantage of Ink is its compilation process. Scripts authored in the Inky editor are compiled into a clean, structured JSON format . This JSON output is lightweight and easily parsed by a runtime engine, making it highly suitable for resource-constrained mobile environments. For iOS integration, a community-developed wrapper, `ink-iOS` , facilitates the use of the Ink runtime within a native application . This wrapper leverages iOS's `JSContext` to execute the JavaScript version of the Ink runtime (`ink.js`), providing a bridge between the compiled JSON story file and the native Swift or Objective-C application code . This approach allows the app to load a pre-authored conversation tree (e.g., `story.json`) and programmatically step through the narrative, presenting the text and choices to the user and handling their selections to progress the story. The `ink-iOS` wrapper provides a simple API, with methods like `canContinue` , `continueStory` , and `chooseChoiceIndex` , which abstract the underlying JavaScript execution and allow for seamless integration into the app's user interface . This architecture ensures that the complex branching logic is handled efficiently by the lightweight Ink engine, while the native iOS app focuses on rendering the UI and managing user interactions.

While other narrative design tools were considered, they were ultimately deemed less suitable for this specific project. **Twine**, for instance, is a popular tool for creating interactive fiction, but it is primarily designed for web-based output and lacks a native iOS application or a straightforward path to integration within a native app . Using Twine would likely require embedding a web view, which can introduce performance overhead and complicate the communication between the narrative and the app's core features (like location services or phone calls). Furthermore, Twine's browser-based nature presents challenges for offline storage and data persistence on iOS, as its data can be erased by the system if not explicitly saved or if the user clears their browser data . Another alternative, **Yarn Spinner**, is a powerful dialogue system tightly integrated with the Unity game engine . While it excels in game development, its reliance on Unity makes it a poor fit for a native iOS application built with Swift and UIKit or SwiftUI. The overhead of embedding the Unity runtime would far exceed the 500MB budget and introduce unnecessary complexity. Therefore, Ink's combination of a mature, purpose-built language, a proven track record, an open-source ecosystem, and a viable path for native iOS integration makes it the definitive choice for implementing the app's conversational decision trees.

1.2.2. Implementing a Conversational UI with MessageKit

To present the branching narratives authored in Ink to the user, a robust and flexible chat user interface (UI) is required. For the iOS platform, **MessageKit** is the recommended library for this purpose. It is a highly-regarded, Swift-native library with over 6,000 stars on GitHub, indicating strong community support and reliability. MessageKit is built on a `UICollectionView` architecture, which provides excellent performance and scalability, even with long and complex conversation histories. This is a critical consideration for an emergency app, where the conversation flow could be lengthy and detailed. The library's architecture is designed to be highly customizable, supporting not only standard text messages but also a wide array of rich content types, including photos, locations, and custom message cells. This flexibility is essential for an emergency preparedness app, which may need to display images (e.g., for first aid illustrations), maps (e.g., for AED locations), or interactive elements like action buttons and rich cards within the conversation stream. The ability to create custom message types allows the app to seamlessly integrate the Ink narrative's choices as tappable buttons or other interactive UI elements, providing a fluid and intuitive user experience that goes beyond simple text-based menus.

The integration of MessageKit with the Ink narrative engine, managed through the `ink-iOS` wrapper, follows a clear and logical pattern. The app's primary view controller, which hosts the MessageKit UI, would instantiate an `InkStory` object with the compiled JSON file . The main interaction loop would involve checking `inkStory.canContinue` and, if true, calling `inkStory.continueStory` to retrieve the next block of narrative text. This text would then be formatted as a `Message` object and inserted into the MessageKit `MessagesCollectionView` . When the narrative presents choices, the `inkStory.currentChoices` property provides an array of dictionaries, each containing the choice's display text and a unique index . The app can then dynamically generate a series of custom message cells, each containing a button for a choice. When the user taps a button, the app calls `inkStory.chooseChoiceIndex` with the corresponding index, which advances the story down the selected branch. This tight integration ensures that the UI remains perfectly synchronized with the state of the Ink story, providing a responsive and engaging conversational experience.

For the Android version of the app, a direct equivalent to MessageKit is not available. However, the same UI patterns can be achieved by building a custom implementation using a `RecyclerView` . This approach provides the necessary flexibility to create a chat-like interface with custom view holders for different message types (text, images, buttons). Alternatively, a commercial SDK like **Stream Chat SDK** could be considered. The Stream SDK offers comprehensive offline support, using SQLite and Room for

persistence, and provides pre-built UI components for both XML and Jetpack Compose . While it is a more heavyweight solution than a custom `RecyclerView` , it could accelerate development by providing a full-featured, production-ready chat infrastructure. The choice between a custom implementation and a third-party SDK on Android would depend on the project's specific timeline and resource constraints. Regardless of the platform, the goal is to create a UI that feels less like a rigid, robotic "phone tree" and more like a natural, helpful conversation, a design philosophy that will be explored in the next section.

1.2.3. Design Patterns for a Natural, Non-Robotic User Flow

To elevate the user experience from a simple, robotic decision tree to a genuinely helpful and conversational interface, several key design patterns must be implemented. The primary goal is to make the interaction feel less like navigating a rigid, pre-programmed menu and more like receiving guidance from a calm, knowledgeable assistant. One of the most effective techniques is the use of **variable insertion** and **acknowledgment text**. By storing key pieces of information from the user's earlier inputs as variables within the Ink script, the app can reference them later. For example, if a user states they are feeling cold and wet, the app can later respond with, "Sarah, I understand you're feeling cold and wet. Let's focus on getting you warm." This simple act of acknowledging and reusing information makes the interaction feel personal and attentive, a stark contrast to the impersonal nature of traditional IVR systems.

Another critical pattern is the use of acknowledgment and validation before posing a question or presenting a new piece of information. Instead of abruptly asking, "Are you bleeding?", the app should first acknowledge the user's previous statement with empathetic text like, "I understand. That sounds like a very difficult situation." This brief interlude serves to validate the user's experience and creates a more compassionate tone, making the subsequent questions feel less like an interrogation and more like a caring inquiry . Furthermore, the presentation of choices should be managed carefully to avoid the "phone tree" feel. Rather than displaying all possible options at once, the UI can present them sequentially or use conditional logic to only show relevant choices based on the current context. This prevents the user from being overwhelmed with a long list of buttons and keeps the conversation focused. Finally, the ability for the user to interrupt the flow or go back to a previous step is a hallmark of a truly conversational interface. The app should be designed to handle these deviations gracefully, allowing the user to explore information or correct a mistake without having to restart the entire process from the beginning.

1.3. Critical Pre-Development Decisions

1.3.1. Resolving NHS Content Licensing and Geographic Restrictions

A significant architectural and legal challenge identified during the research phase pertains to the use of content from the UK's National Health Service (NHS). While the NHS provides a Content API that appears to be a rich source of medical information, its licensing terms present a major hurdle for an application designed for global, offline use. The NHS Syndication License imposes a strict **UK geographic restriction on users**, meaning that content can only be accessed and cached by individuals who can be verified as being physically located within the United Kingdom or who possess a UK-based account . This requirement fundamentally conflicts with the core value proposition of an emergency preparedness app, which must be fully functional regardless of the user's location, especially during international travel or in scenarios where location verification is impossible due to a lack of connectivity. The license permits offline caching of content, but this is only permissible for verified UK users. Therefore, deploying this content in an app that is "download once, use anywhere" would constitute a breach of the licensing agreement. This issue is a potential dealbreaker that must be resolved before any investment is made in developing pipelines for NHS content. The development team faces a critical decision with three primary paths: accept the UK-only restriction, which simplifies development but severely limits the app's market and utility; find alternative, freely redistributable medical sources that do not have such geographic limitations; or implement a complex and potentially unreliable location verification system, which would add significant development overhead and could fail the "airgapped" requirement if the verification process requires network access. The resolution of this issue is paramount and will dictate the entire content strategy for the medical reference component of the application.

The implications of the NHS licensing restriction extend beyond mere legal compliance and touch upon the very design philosophy of the application. An emergency app is, by its nature, a tool for unforeseen circumstances, which often occur outside one's home country or in areas with limited infrastructure. Tying the app's core medical reference functionality to a specific geographic region undermines its reliability and preparedness value. For instance, a UK citizen traveling abroad who encounters a medical emergency would be unable to access the app's NHS-sourced guidance, rendering a key feature useless at the most critical moment. This creates a fragmented and unreliable user experience. The alternative paths each carry their own trade-offs. Finding alternative

medical sources, such as content from OpenWHO or WikiDoc, would require a significant research effort to ensure the quality, accuracy, and licensing terms of the new data. It would also necessitate rebuilding the content acquisition and processing pipelines. Implementing a location verification system, while technically feasible, introduces a new layer of complexity and potential points of failure. Such a system would need to be robust enough to handle various edge cases (e.g., GPS spoofing, VPNs, border regions) and would likely require an initial network connection for verification, contradicting the "airgapped" ideal. Therefore, the decision is not merely a technical one but a strategic choice about the app's fundamental purpose and target audience. A thorough evaluation of the available alternatives and a clear decision on the path forward are essential prerequisites for any further development.

1.3.2. Prototyping and Validating the Natural Language Lookup UX

A core assumption of the proposed architecture is that a lightweight, on-device intent classification model can provide a sufficiently intelligent and satisfying user experience, effectively routing user queries to the correct offline information. To mitigate the risk of building a full system based on this unproven thesis, it is imperative to conduct a rapid prototyping and validation exercise. This "weekend spike" would focus on testing the interaction model before committing to the full development pipeline. The first step involves defining a representative set of 20-30 user intents that cover the breadth of potential emergency queries, from simple requests like "how to treat a burn" to more complex, conversational inputs such as "I'm cold and wet, what should I do?". Once these intents are defined, a synthetic dataset of example phrases for each intent must be created. This dataset will be used to train a small-scale MLTextClassifier model using Apple's CreateML framework. The goal of this prototype is not to achieve perfect accuracy but to test the fundamental interaction loop: user speaks, app classifies, app retrieves relevant content. A critical part of this validation is to test the classifier's ability to handle ambiguous or non-literal queries. For example, the phrase "I'm cold and wet" should ideally be classified as an intent related to hypothermia or exposure, rather than being treated as a literal query about being cold and wet. This test will determine if the classification feels "smart enough" to be genuinely helpful or if it will be perceived as frustratingly literal and brittle, forcing users to learn a specific set of keywords rather than speaking naturally. The outcome of this spike will be a go/no-go decision on the primary UX thesis, potentially saving significant development time if the approach proves unsatisfactory.

The validation process must be rigorous and user-centric to yield meaningful results. It is not enough for the classifier to achieve high accuracy on a curated test set; it must perform well with the kind of varied, often panicked, and grammatically imperfect language that people use in real emergencies. The prototype should therefore include a simple interface that allows for open-ended text input, simulating a real-world chat interaction. Testers should be encouraged to ask questions in their own words, without being given a list of approved commands. The system's responses, even if they are just placeholders indicating which intent was triggered, should be analyzed for their relevance and usefulness. For instance, if a user asks, "My child swallowed something, what do I do?", the system must correctly identify this as a "poisoning" or "choking" intent, even though the query does not contain those specific keywords. This requires the model to understand semantic relationships and context, a challenge for smaller, traditional ML models. The prototype should also test the system's ability to handle multi-turn conversations. Can it maintain context from one query to the next? If a user first asks about treating a cut and then follows up with "what if it's bleeding a lot?", can the system understand that the second query is related to the first? The answers to these questions will provide invaluable insight into the viability of the natural language lookup approach and will inform the decision of whether to proceed with the planned architecture or to pivot to a more sophisticated, albeit more resource-intensive, solution.

1.3.3. Authoring and Testing a Complete Conversation Tree

Beyond simple information retrieval, the application aims to provide interactive, step-by-step guidance for critical emergency scenarios. This functionality is to be implemented using pre-authored conversation trees, with the Ink scripting language from Inkle identified as the most mature and suitable format . Before investing in the complex data pipeline required to integrate these trees into the app, it is crucial to first validate the user experience of the conversation trees themselves. This involves authoring a single, complete conversation tree for a critical scenario, such as medical triage for a breathing emergency, from start to finish. The focus of this exercise is purely on the branching logic, narrative flow, and conversational feel of the script, independent of any technical implementation. The goal is to determine if the resulting interaction feels genuinely conversational and supportive, or if it comes across as a rigid, robotic "phone tree" that would be frustrating or stressful to use in a real emergency. The script should be designed to be as natural as possible, incorporating techniques such as variable insertion (referencing earlier user choices), acknowledgment text to show the system is listening ("I understand, that sounds very

difficult"), and conditional text variants based on the user's accumulated state. The tree should be manually walked through multiple times, simulating different user paths and emotional states, to identify any points of friction, confusion, or unnatural phrasing. This hands-on testing will provide a clear qualitative assessment of the conversation tree approach and will highlight any necessary adjustments to the writing style or structure before the content creation process is scaled up.

The process of authoring and testing a complete conversation tree serves as a critical reality check for the project's conversational UI ambitions. It is one thing to theorize about creating a natural, supportive dialogue system, and another to actually write and experience one. The test script should be designed to push the boundaries of the format, exploring how to handle interruptions, digressions, and user anxiety. For example, the script should include mechanisms for allowing the user to ask "why?" at any point, or to request that the instructions be repeated. It should also be able to handle situations where the user provides information that is not directly relevant to the current question. The writing style is paramount; the language must be clear, concise, and reassuring, avoiding medical jargon wherever possible. The use of typing indicators and slight delays (e.g., 200–400ms) can also be simulated during the manual walkthrough to see if they enhance the feeling of a real-time conversation. The feedback from this exercise will be invaluable. It may reveal that the Ink format is perfectly suited for the task, or it may uncover limitations that necessitate a more flexible, state-machine-based approach. It could also lead to the development of a specific style guide for authoring the conversation trees, ensuring consistency and quality across all scenarios. Ultimately, this pre-development step is about validating the human-centered design of the app's most critical feature, ensuring that the technology serves the user's needs in the most effective and empathetic way possible.

2. On-Device Intelligence and Performance

2.1. Intent Classification and Entity Extraction

2.1.1. Primary Approach: Apple's NaturalLanguage Framework (MLTextClassifier)

For the core task of intent classification, the primary and recommended approach is to leverage **Apple's NaturalLanguage framework**, specifically the `MLTextClassifier` class. This framework is part of Apple's core ML stack and is highly optimized for on-device performance, making it an ideal choice for a resource-constrained mobile application. The `MLTextClassifier` is designed to categorize text into a predefined set of labels, which perfectly aligns with the app's need to route user queries to the

correct emergency protocol. The models trained with this framework are exceptionally lightweight, typically ranging from **1–10MB** in size, and can achieve **99%+ accuracy** on well–defined intent sets of 5–50 categories. This small footprint is a significant advantage, as it leaves more room in the 500MB budget for other critical content like maps and reference documents. Furthermore, the inference latency is extremely low, with benchmarks showing performance of **3–10ms** on the A18 Pro's Neural Engine. This near–instantaneous response time is crucial for maintaining a fluid and responsive conversational interface, ensuring that users are not left waiting for the app to process their input.

The process of creating a custom text classifier is streamlined through the use of **CreateML**, Apple's machine learning model training tool. CreateML provides a simple, visual interface for training models without requiring deep expertise in machine learning. The developer can provide a training dataset in the form of a JSON file, where each entry consists of a text sample and its corresponding label (intent). For example, a sample entry might be `{"text": "I am having chest pains", "label": "cardiac_arrest"}`. CreateML then handles the process of training and validating the model, outputting a `.mlmodel` file that can be directly integrated into the iOS application. This workflow allows for rapid iteration and refinement of the classifier, enabling the development team to easily update the model as new data becomes available or as the scope of the app's intents expands. The combination of a lightweight, high–performance framework and a user–friendly training tool makes the `MLTextClassifier` the clear first choice for implementing the app's natural language understanding capabilities.

2.1.2. Alternative Approach: Evaluating Small Language Models (MobileLLM)

While the `MLTextClassifier` is the recommended primary approach, there may be scenarios where a more complex understanding of user intent is required. In such cases, a small, on–device Large Language Model (LLM) could be considered as an alternative. For applications with a sub–500MB budget, **MobileLLM–350M from Meta** represents the most viable option. After applying INT4 quantization, this model has a size of approximately **200MB**, which is still within the acceptable range for a mobile application. Meta's research has shown that the MobileLLM architecture, with its deep and thin design, is particularly well–suited for on–device API calling tasks and can achieve accuracy on par with much larger models like LLaMA–2 7B on intent classification and slot–filling benchmarks. This suggests that MobileLLM could provide a more nuanced and flexible understanding of user queries than a traditional text classifier.

However, the use of MobileLLM comes with several significant trade-offs that must be carefully considered. The most notable is its **FAIR Non-Commercial Research license**, which creates complications for commercial deployment. Any app using this model would need to be distributed for free, which may not align with the project's long-term goals. Additionally, while 200MB is manageable, it is still a substantial portion of the 500MB budget and could limit the amount of other content that can be included. The computational overhead of running a 350M parameter model is also significantly higher than that of a 1-10MB text classifier, which could have a more noticeable impact on battery life and device performance. Another model, **TinyLlama-1.1B**, has been evaluated but is not recommended. Even after INT4 quantization, its size of approximately **600MB** exceeds the entire budget for the application on its own. Therefore, while MobileLLM-350M presents a technically interesting alternative for more advanced intent understanding, its licensing restrictions and larger resource footprint make it a less practical choice than the `MLTextClassifier` for the primary implementation.

2.1.3. CoreML Conversion and Optimization for iOS

If a decision is made to use a model other than the native `MLTextClassifier`, such as a small LLM like MobileLLM or a custom transformer model, it must be converted to the **CoreML format** to run efficiently on iOS devices. The primary tool for this conversion is `coremltools`, and it is recommended to use **version 8.0 or later** to ensure compatibility with the latest iOS features and model architectures. The conversion process can be complex and requires careful attention to several potential pitfalls. One common issue is the handling of **flexible input shapes**. While CoreML supports models with dynamic input sizes, these models will only run on the CPU, which is significantly slower than the Neural Engine. To take full advantage of the A18 Pro's hardware acceleration, it is crucial to use **fixed or enumerated input sizes** during the conversion process. This allows the model to be compiled and optimized for the Neural Engine, resulting in much faster inference times.

Another critical area to address during conversion is the handling of **attention masks**. Transformer-based models rely on attention mechanisms to process sequences, and the way the attention mask is implemented can sometimes lead to cryptic errors during the CoreML conversion. It is essential to carefully review the model's architecture and ensure that the attention mask is correctly defined and passed to the conversion function. Furthermore, the **tokenizer** used to convert text into numerical tokens for the model is often a separate component that is not included in the model file itself. For

iOS integration, the tokenizer must be bundled with the app as a separate resource. The `swift-transformers` library provides a convenient way to load and use popular tokenizers, such as those from the Hugging Face ecosystem, within a Swift application. The typical conversion pipeline for a model from Hugging Face would be to first export it to the **ONNX format**, and then use `coremltools` to convert the ONNX model to CoreML. Finally, to further reduce the model's size and improve its performance, **post-training quantization** should be applied. The `quantize_weights` function in `coremltools` can be used to apply **INT4 quantization** with a block size of 32, which can significantly reduce the model's memory footprint with minimal impact on accuracy.

2.2. Battery and Performance Optimization

2.2.1. A18 Pro Neural Engine Power Consumption Analysis

A key concern for any on-device machine learning application is its impact on battery life. However, for the intermittent classification workload of the emergency app, the power consumption on the A18 Pro is expected to be negligible. Apple's own research provides concrete data on the power efficiency of its Neural Engine. For a model like DistilBERT, which is similar in complexity to the `MLTextClassifier`, inference can be performed in **3.47ms at a power draw of 0.454W** in a high-performance mode, or in **9.44ms at a much lower 0.072W** in a more energy-efficient mode. Given that the app will likely perform at most a few thousand classifications per day, the total energy consumption will be minimal. A rough calculation based on these figures suggests that **1,000 inferences would consume approximately 1-2 mAh** of battery. On the iPhone 16 Pro's 3,577 mAh battery, this represents a fraction of a percent of the total capacity, making the power impact effectively invisible to the user. This low power consumption is a direct result of the specialized hardware in the Neural Engine, which is designed to perform the matrix multiplications and other operations common in machine learning far more efficiently than the general-purpose CPU or GPU.

2.2.2. Memory Management with Memory-Mapped Model Loading

To ensure that the app remains responsive and does not consume excessive RAM, it is essential to implement efficient memory management techniques for loading the machine learning model. The recommended approach is to use **memory-mapped model loading**. Instead of loading the entire model file into memory at once, this technique maps the file into the app's virtual address space. The operating system then lazily loads the parts of the file into physical RAM as they are needed. This has several

advantages. First, it reduces the app's initial memory footprint, as the entire model is not loaded into RAM immediately. This is particularly important for preventing the app from being "jettisoned" or terminated by the system when it is in the background, a common occurrence for apps that use a large amount of memory. Second, it allows for more efficient sharing of memory between multiple processes that might be using the same model file. Both CoreML and ONNX Runtime Mobile support memory-mapped loading, and it should be the default strategy for managing the resident RAM of the intent classification model. The target for the total memory usage of the model should be **under 500MB** to ensure reliable operation on a wide range of devices.

2.2.3. Android Parity with ONNX Runtime Mobile

To ensure feature parity between the iOS and Android versions of the app, a similar on-device machine learning solution must be implemented for the Android platform. The recommended tool for this is **ONNX Runtime Mobile**, which provides a high-performance inference engine for ONNX models on Android devices. ONNX (Open Neural Network Exchange) is an open format for representing machine learning models, which allows for a model to be trained in one framework (e.g., PyTorch) and then deployed in another (e.g., ONNX Runtime). This provides a high degree of flexibility and avoids vendor lock-in. For hardware acceleration on Android, ONNX Runtime Mobile can be configured to use the **NNAPI (Neural Networks API) delegate**. NNAPI is Android's equivalent to CoreML, providing a standard interface for running computationally intensive machine learning operations on the device's dedicated AI processors, such as the Qualcomm Hexagon DSP or Samsung's NPU. By using ONNX Runtime Mobile with the NNAPI delegate, the Android app can achieve inference performance that is comparable to the CoreML-powered iOS app, ensuring a consistent and responsive user experience across both platforms. Another option to consider is **MediaPipe's LLM Inference API**, which supports models like Gemma 2B and custom models, but it requires the use of the larger `.task` format, which may not be suitable for the app's strict size constraints.

3. Offline Data Architecture and Storage

3.1. Primary Database: SQLite with FTS5

3.1.1. Configuration for Maximum Read-Heavy Performance

The foundation of the app's offline data architecture is **SQLite**, a self-contained, serverless database engine that is a perfect fit for mobile applications. For an app that

is primarily focused on reading and searching through a large amount of reference data, SQLite's performance is excellent. To maximize its performance for this read-heavy workload, several key configuration pragmas should be set. First, the **journal mode should be set to WAL (Write-Ahead Logging)** . This is a modern journaling mode that allows for concurrent reading and writing, which can significantly improve performance in multi-threaded applications. Second, the **synchronous mode should be set to NORMAL**. This provides a good balance between data integrity and performance, ensuring that the database is not overly slowed down by excessive disk synchronization operations. Finally, the **cache size should be increased** to allow more data to be held in memory. A cache size of **-64000** (which corresponds to 64MB) is a reasonable setting for a modern iPhone, and it will reduce the number of disk reads required for common queries, leading to faster response times.

3.1.2. Implementing Full-Text Search with FTS5 Contentless Indexes

To enable fast and powerful full-text search across the entire knowledge base, the app will use **SQLite's FTS5 extension**. FTS5 is a high-performance full-text search engine that is built into SQLite, and it is the recommended choice for this type of application. A key optimization for the emergency app is the use of **contentless indexes with** `detail=none` . A standard FTS index stores a copy of the original text, which can significantly increase the size of the database. A contentless index, on the other hand, only stores the index data and a reference to the original document, which can reduce the storage overhead to approximately **20-30%** of the source text size. The `detail=none` option further reduces the size of the index by not storing position information, which is not required for simple keyword searches. The FTS5 index should be configured to use the **Porter stemming algorithm** for English content. Stemming improves search recall by reducing words to their root form, so a search for "running" will also match documents containing "run" or "ran". This combination of a contentless index and stemming provides a powerful and efficient full-text search capability that is well-suited to the app's resource constraints.

3.1.3. Handling Locale-Specific Content with Jurisdiction Fallbacks

To support a global user base and provide information that is relevant to the user's location, the app's content database will be designed to handle locale-specific data. This will be achieved by adding a `jurisdiction` column to the main content tables. This column will store a code indicating the geographic region to which the content applies, such as "UK" for the United Kingdom, "EU" for the European Union, or "INTL" for international content. The application logic will then implement a **priority-based**

fallback system. When a user performs a search, the app will first query for content that matches their current jurisdiction. If no results are found, it will then query for content from a secondary jurisdiction (e.g., EU), and finally, it will fall back to the international content. This approach allows the app to provide highly specific information, such as local laws or emergency numbers, while still ensuring that a baseline level of information is always available. This jurisdiction-based system provides a flexible and scalable way to manage a diverse and geographically distributed content library.

3.2. Advanced Search and Data Optimization

3.2.1. Semantic Search with sqlite-vec and Pre-Computed Embeddings

To provide a more intelligent search experience that goes beyond simple keyword matching, the app will incorporate **semantic search**. This allows the app to understand the meaning and context of a user's query, enabling it to find relevant information even if the query does not contain the exact keywords. The foundation of semantic search is the use of **vector embeddings**, which are numerical representations of text that capture its semantic meaning. To implement this on-device, the app will use the `sqlite-vec` **extension**, which provides vector search capabilities directly within SQLite. To avoid the computational overhead of generating embeddings in real-time on the user's device, the recommended approach is to **pre-compute the embeddings at build time**. The `all-MiniLM-L6-v2` model, a lightweight and efficient sentence-transformer, will be used to generate 384-dimensional embeddings for all the text chunks in the knowledge base. These pre-computed vectors will then be stored as BLOBs in the SQLite database. At runtime, when a user enters a query, the app will generate an embedding for the query text and then use `sqlite-vec` to perform a fast similarity search against the pre-computed embeddings, finding the most semantically relevant content. The storage overhead for these embeddings is modest, with each 384-dimensional vector consuming approximately **1.5KB**, meaning a database of 10,000 documents would only require about **15MB** for the embeddings.

3.2.2. Extreme Compression with Zstandard and Trained Dictionaries

Given the strict 500MB budget for the entire application, achieving extreme compression of the offline content is a fundamental requirement. While standard compression algorithms can provide some reduction in file size, the use of **Zstandard (ZSTD)** in conjunction with **trained dictionaries** offers a pathway to dramatically higher compression ratios, particularly for the homogeneous text data typical of legal

documents, medical guides, and survival manuals . Zstandard is a modern, fast compression algorithm that provides a wide range of compression levels. For an application where data is pre-processed and bundled at build time, the highest compression levels can be used without impacting the user's runtime experience. The real breakthrough, however, comes from using trained dictionaries. This technique involves analyzing a large sample of the data to be compressed and creating a custom dictionary that captures the most common patterns, words, and phrases. When this dictionary is then used to compress the full dataset, the algorithm can achieve remarkable results.

The impact of this approach is profound. On similar medical or legal documents, standard compression might yield a modest 10-20% size reduction. In contrast, using Zstandard with a domain-trained dictionary can achieve compression ratios of **50:1 to 100:1** . This means a 100MB collection of raw text could be compressed down to just 1-2MB. To implement this, the build process would involve training separate dictionaries for each major content category (e.g., a 32-112KB dictionary for medical texts, another for legal documents, and a third for survival guides). This targeted approach ensures that the dictionary is highly optimized for the specific vocabulary and structure of each content type. The resulting compressed files are then bundled with the application. At runtime, the app would load the appropriate dictionary into memory and use it to decompress the content on demand. While this adds a small amount of complexity to the build pipeline and a minor memory overhead at runtime (for the dictionary itself), the payoff in storage savings is immense. This technique is the key to fitting a comprehensive, multi-domain knowledge base within the tight constraints of a mobile application, making it a cornerstone of the project's data architecture.

3.2.3. Zero-Copy Data Access with FlatBuffers

For certain types of read-only reference data, such as the index for the conversation trees or other structured data, the use of **FlatBuffers** can provide a significant performance advantage over traditional serialization formats like JSON. FlatBuffers is an efficient cross-platform serialization library that allows for **zero-copy access** to data. This means that the data can be read directly from the memory-mapped file without the need for a separate deserialization step, which can be a time-consuming process for large or complex objects. The performance difference is substantial, with FlatBuffers achieving an access time of approximately **77ns per access**, compared to around 500ns for JSON parsing. This speed is critical for an app that needs to be highly responsive. The implementation would involve defining the data schema in a

`.fbs` file, using the FlatBuffers compiler to generate the necessary code for Swift, and then serializing the data at build time. The resulting binary file can then be memory-mapped and accessed directly by the app at runtime, providing a highly efficient and performant way to handle structured, read-only data.

3.3. iOS Storage and File Management

3.3.1. Choosing the Correct File Directory (Application Support vs. Caches)

A critical aspect of designing a robust offline-first application for iOS is the strategic selection of file system locations for storing different types of data. The iOS file system has specific directories with distinct behaviors regarding backup, persistence, and purging, and choosing the wrong one can lead to data loss, excessive iCloud storage usage, or a poor user experience. For the emergency preparedness app, a two-tiered storage strategy is recommended to manage the various types of offline content effectively . The first tier, for **essential, non-replaceable data**, should be stored in the **`Library/Application Support/`** directory. This location is the ideal choice for the core emergency content, including the pre-processed SQLite databases, the Ink conversation trees, and the Zstandard-compressed reference documents. Files placed in `Application Support` are guaranteed to be backed up by iCloud (if the user has it enabled), are not purged by the system during low-storage situations, and persist across app updates. This ensures that the life-saving information the app provides is always available to the user and is not lost due to system maintenance or device upgrades.

The second tier of the storage strategy is for **downloadable or cacheable content** that can be re-fetched from a server if necessary. This includes items like map tile updates, non-critical reference documents, or other large assets that are not part of the core emergency functionality. This type of data should be stored in the **`Library/Caches/`** directory. The primary advantage of using `Caches` is that files stored here are **not** backed up to iCloud, which prevents the app from consuming the user's valuable cloud storage quota with large, easily replaceable files. While the system may purge the contents of the `Caches` directory when the device is under low storage pressure, this is an acceptable trade-off for data that can be downloaded again. This two-tiered approach provides a balanced and efficient storage architecture: it guarantees the persistence of critical, offline-first data while managing the device's storage footprint responsibly by treating less critical, downloadable content as transient cache.

3.3.2. Ensuring Offline Accessibility with File Protection Levels

In an emergency situation, a user's device may be locked, and they may not have the time or ability to unlock it to access critical information. Therefore, ensuring that the app's offline data is accessible even when the device is in a locked state is a crucial design consideration. iOS provides a mechanism for this through its **file protection levels**, which are part of the broader Data Protection API. By default, files created by an app are encrypted and inaccessible when the device is locked. However, for an emergency app, this default behavior is undesirable. The recommended solution is to explicitly set the file protection level to `.completeFileProtectionUntilFirstUserAuthentication` when writing all critical offline data to the file system .

This specific protection level provides a secure yet accessible compromise. It ensures that the files are encrypted and protected until the user unlocks the device for the first time after a reboot. Once the initial unlock has occurred, the files remain decrypted and accessible even if the device locks again. This is the ideal behavior for an emergency app: it protects the user's data from unauthorized access if the device is stolen and has not yet been unlocked, but it allows the legitimate owner to access the app's content from the lock screen (e.g., via a widget or notification) or immediately after a quick unlock. Implementing this requires a small but important code change when writing files. For example, in Swift, the `write(to:options:)` method should be called with the `.completeFileProtectionUntilFirstUserAuthentication` option. This ensures that all the vital information stored in the `Library/Application Support/` directory, such as medical protocols and legal rights, is available at a moment's notice, which could be the difference between life and death in a true emergency.

3.3.3. Managing Large Datasets with the Background Assets Framework

For an application that relies on a substantial amount of offline data, the initial download and subsequent update process must be handled gracefully to avoid a poor first-time user experience. The traditional approach on iOS, **On-Demand Resources (ODR)**, is now considered a legacy technology, and Apple explicitly recommends that developers migrate to the newer **Background Assets framework**, introduced in iOS 16.1 . Background Assets is a superior solution for managing large downloads because it is designed to begin downloading essential content *before* the user launches the app for the first time. This is a game-changer for an emergency app, as it ensures that the core data is already on the device when the user first opens it, eliminating a potentially long and frustrating wait during a critical moment. The framework is configured through keys in the app's `Info.plist` , such as `BAManifestURL` , which points to a

manifest file on a developer-controlled CDN, and `BAEssentialDownloadAllowance`, which specifies the size of the critical content that must be downloaded before the app can launch.

This "essential" download feature is particularly powerful. It effectively extends the App Store's installation progress bar to include the download of the most critical emergency data. The app will not fully launch until this essential content is present, guaranteeing a baseline level of functionality from the very first interaction. For less critical or larger datasets, such as detailed map tiles for the entire UK, the Background Assets framework can be used to download these in the background after the app has launched. This ensures the app is usable immediately while progressively enhancing its offline capabilities. For any runtime downloads that occur after the initial installation, the app should utilize **NSURLSession background sessions**. This system-level service handles downloads even if the app is suspended or terminated, ensuring reliability and resumability. By combining the pre-launch capabilities of Background Assets with the robustness of background URL sessions, the app can deliver a seamless and reliable experience for acquiring and updating its large offline datasets, all while respecting the user's time and device resources.

4. Content Strategy and Data Sources

4.1. UK Open Government Licence (OGL) Datasets

4.1.1. Legal Framework: Legislation from legislation.gov.uk

A cornerstone of the app's legal reference section will be the comprehensive collection of UK legislation available from legislation.gov.uk. This official government website provides access to all primary and secondary legislation, including the critical Police and Evidence Act 1984 (PACE) codes, all of which are published under the Open Government Licence (OGL). This means the content can be freely downloaded, stored offline, and redistributed within the app without requiring any special permissions or registration, as long as the required attribution is provided. The availability of this data is a significant advantage, as it allows the app to offer users authoritative and up-to-date information on their legal rights and the powers of emergency services in various situations. The legislation is accessible through a RESTful API that can return data in multiple formats, including XML, RDF, and HTML, providing flexibility in how the content is ingested and processed for the mobile application.

The technical implementation will involve creating a build-time pipeline to fetch and parse the relevant legislative documents. For example, the entire Civil Contingencies Act 2004, which forms the bedrock of the UK's emergency planning framework, can be downloaded and processed into a structured format for the app's database. The XML format is particularly useful for this purpose, as it allows for the extraction of structured data such as section numbers, headings, and the body text of the law. This structured data can then be indexed using SQLite's FTS5 extension to enable powerful full-text search capabilities, allowing users to quickly find specific information within the legal texts. The estimated size of a compressed bundle containing key emergency legislation, such as the Civil Contingencies Act and the PACE codes, is approximately 10-20MB, which is well within the app's overall 500MB budget. By including this legal framework, the app can empower users with the knowledge of their rights and the official procedures that govern emergency response in the UK.

4.1.2. Mapping Data: OS OpenData vs. OpenStreetMap

For the app's offline mapping capabilities, a critical decision must be made between using the UK's official Ordnance Survey (OS) OpenData products or the community-driven OpenStreetMap (OSM) dataset. Both options have distinct advantages and are available under licenses that permit free redistribution, making them suitable for the project. OS OpenData provides a range of high-quality, authoritative datasets, including OS Open Zoomstack for vector tiles, OS Open Roads for the road network, and Code-Point Open for postcode centroids. The primary advantage of OS data is its official status and high level of accuracy, which is crucial for navigation and location-based services in an emergency. The use of OS data requires attribution, typically in the format: "Contains OS data © Crown Copyright [and database right] [year]". The estimated size for a comprehensive UK vector tile dataset (zoom levels 0-14) from OS OpenData is between 500MB and 1GB, which could consume a significant portion of the app's total budget.

On the other hand, OpenStreetMap offers a highly detailed and globally consistent dataset that is continuously updated by a large community of contributors. While it may lack the official stamp of approval that comes with OS data, its level of detail, particularly for pedestrian paths, cycle routes, and points of interest (POIs), is often superior. OSM data can be processed into vector tiles using tools like OpenMapTiles, which provides a ready-made solution for creating offline map packages. The licensing for OSM-derived data, typically a combination of BSD and CC-BY, is also very permissive and well-suited for this project. A UK-wide OSM vector tile dataset at a

reduced zoom level (e.g., z0–12) could be packaged within a 150–300MB footprint, leaving more room in the budget for other critical content like medical and legal reference materials. Given the strict 500MB budget, using a reduced-zoom OSM dataset appears to be the more pragmatic choice, providing a good balance of detail, flexibility, and size efficiency.

4.1.3. Environmental Data: Environment Agency Flood Information

To provide users with crucial information about environmental risks, the app will incorporate flood data from the Environment Agency (EA). This data is made available under the Open Government Licence (OGL) via the environment.data.gov.uk/flood-monitoring/ API, which requires no registration and allows for free redistribution. This makes it an ideal source for building an offline database of flood risk information for the UK. The API provides access to both static and real-time data. For the purposes of an offline-first, airgapped app, the primary focus will be on the static flood area definitions. This data can be downloaded and pre-packaged into the app's SQLite database, allowing users to check their flood risk even when they have no internet connectivity. This is a critical feature, as it enables proactive planning and preparation before a potential flood event.

The real-time flood warnings and alerts provided by the EA API, while valuable, would require an online connection to fetch and are therefore outside the scope of the core airgapped functionality. However, the architecture could be designed to allow for an optional online mode where this real-time data could be integrated to provide enhanced situational awareness when connectivity is available. The static flood area data is relatively compact, with an estimated size of around 20MB when compressed, making it a very cost-effective addition to the app's content library. By including this data, the app can provide users with a vital tool for understanding their local environmental risks and taking appropriate action to protect themselves and their property. The integration of this data will involve downloading the relevant shapefiles or data feeds from the EA, processing them to extract the key information (e.g., flood zone, risk level, location), and storing this in a searchable format within the app's database.

4.2. Medical and Health Information

4.2.1. Navigating NHS Content Licensing and Geographic Restrictions

Incorporating authoritative medical information from the UK's National Health Service (NHS) presents a significant legal and technical challenge due to the restrictive nature of its licensing. The NHS Website Content API, which provides access to valuable resources like the Health A–Z and Medicines A–Z, operates under a Syndication License that explicitly requires users to be geographically located within the UK. This geographic restriction is a major hurdle for an app designed for universal, offline use, as it would prevent non–UK residents or travelers from accessing critical medical information. While the license does permit the caching of content for offline use, this is only allowed for users who have been verified as being in the UK. This creates a fundamental conflict with the app's core value proposition of being a reliable, airgapped emergency tool.

The development team faces a critical decision on how to proceed. The first option is to accept the UK–only limitation, which would simplify the technical implementation but severely restrict the app's global applicability and usefulness for travelers. The second option is to seek alternative sources of medical information that are licensed for global redistribution. This could involve exploring content from international health organizations like the World Health Organization (WHO), or from open–source medical projects like WikiDoc, which may have more permissive licensing terms. The third, and most complex, option is to implement a location verification system within the app. However, this approach is fraught with difficulties. It would add significant complexity to the app, potentially require an internet connection for the initial verification, and could fail in a true offline emergency scenario. Given these challenges, the most robust and ethically sound path is likely to pursue alternative medical content sources that are explicitly licensed for worldwide, offline use, thereby ensuring the app remains a truly universal emergency preparedness tool.

4.2.2. Accessing NICE Guidelines via the Syndication API

The National Institute for Health and Care Excellence (NICE) provides a comprehensive repository of clinical guidelines, standards, and evidence–based recommendations that would be an invaluable asset to an emergency preparedness app. Access to this content is facilitated through the NICE Syndication Service, a RESTful API that allows for programmatic retrieval of its resources . The service supports multiple data formats, including XML and JSON, which are essential for structured data ingestion and processing within a mobile application. The API is organized hierarchically, allowing developers to navigate through different types of content, such as guidance by date, by program (e.g., clinical guidelines, diagnostics guidance), or by a structured taxonomy.

This structure is beneficial for selectively downloading relevant content for an emergency app, such as guidelines on trauma, cardiac events, or infectious diseases. However, utilizing this API is not a simple matter of making HTTP requests. It requires a formal application process to obtain an API key and a license agreement. The license, as previously noted, includes a critical geographic restriction, limiting content access to users within the UK . This is a major hurdle for an app designed for global, offline use. Furthermore, the API does not appear to offer a single "bulk download" endpoint for all guidelines. Instead, a developer would need to write a script to iterate through the various indices and endpoints to download each guideline individually, a process that would need to be managed carefully to respect API rate limits and handle potential errors. The guidelines themselves can be retrieved in several formats, including as complete documents, individual chapters, or as a "structured document" where the content is broken down into discrete XML elements, which is ideal for parsing and integrating into a local database . While the NICE Syndication Service offers a path to high-quality, authoritative medical content, the combination of the licensing restrictions and the lack of a simple bulk download mechanism makes it a challenging and potentially unsuitable primary data source for this project's specific requirements.

The technical implementation of integrating NICE guidelines via their API involves several steps, each with its own considerations. After obtaining the necessary API key, the first step is to explore the API's structure to identify the most relevant content. The guidance index and taxonomy endpoints are the logical starting points for this discovery process . For example, a developer could query the taxonomy endpoint to find all guidelines related to "emergency medicine" or "cardiovascular conditions." Once a list of relevant guideline URLs is compiled, the next step is to download the content. The API supports content negotiation via the `Accept` header, allowing a client to request either JSON or XML representations. For the purpose of building a structured, searchable database, the XML format is often preferable due to its well-defined schema for elements like chapters and sections. The "structured document" option is particularly useful here, as it provides the full content of a guideline in a single resource, with distinct XML tags for chapters and sections, making it easier to parse and chunk for storage in a local SQLite database . The download process itself would need to be robust, handling network failures and implementing retry logic. The API supports ETAGs and `Last-Modified` headers, which can be used to implement an efficient update mechanism, only downloading guidelines that have changed since the last sync. However, this entire process is predicated on the user being a verified UK resident, a requirement that is difficult to enforce in an offline-first app and

fundamentally limits the app's scope. The alternative of using the API to pre-populate a database at build time and then distributing that database with the app would likely violate the terms of the syndication license, as it would involve redistributing the content to users outside the UK. This legal and logistical complexity makes the NICE API a less-than-ideal solution, despite the high quality of its content.

4.2.3. Alternative Sources: Wikipedia Medical Content via ZIM Files

Given the significant licensing and geographic restrictions associated with official UK health data sources like the NHS and NICE, a highly viable alternative for building a comprehensive, offline medical knowledge base is to leverage the vast repository of medical articles available on Wikipedia. While not a substitute for official clinical guidelines, Wikipedia's medical content is extensive, collaboratively edited, and, most importantly, freely available under a license that permits redistribution and modification. The key to making this content usable in an offline, resource-constrained mobile application is the ZIM file format, a technology developed by the Kiwix project specifically for packaging web content for offline use . The ZIM format is exceptionally well-suited for this purpose. It is a highly compressed archive that can contain thousands of articles and associated media, but it is structured in a way that allows for random access to individual articles without needing to decompress the entire file. This is achieved through a sophisticated internal architecture that uses a directory of entries and pointer lists, enabling fast, direct lookups of content based on a URL or path . For the emergency app, a specific ZIM file containing only medical content, such as the `wikipedia_en_medicine_nopic` archive, could be used. This file, which contains all articles tagged by Wikipedia's WikiProject Medicine but without images, is estimated to be around 200-250MB, a size that fits comfortably within the app's overall 500MB budget. By using the `libzim` library, the app can directly read and search this ZIM file, providing users with instant access to a wealth of medical information, from detailed articles on specific conditions to general first aid guidance, all without requiring an internet connection.

The process of integrating Wikipedia medical content via ZIM files involves several steps, from content selection and extraction to integration into the app's search and display systems. The first step is to select the appropriate ZIM file. The Kiwix project provides a library of pre-built ZIM files for various topics and languages, including the aforementioned `wikipedia_en_medicine_nopic` . If a more customized selection of articles is needed, the `mwoffliner` tool can be used to create a bespoke ZIM file from a list of Wikipedia URLs. This list could be generated using the PetScan API to query

for articles within specific medical categories or those tagged as "vital articles" . Once the ZIM file is selected, the next step is to integrate the `libzim` library into the iOS and Android applications. For iOS, this involves building the `CoreKiwix.xcframework` from the Kiwix build repository. For Android, the `java-libkiwix` AAR bindings can be used. With the library integrated, the app can then use it to search the ZIM file's index and retrieve article content. While the ZIM file provides its own full-text search capabilities, a more powerful approach for the emergency app would be to pre-process the content at build time. This would involve extracting the HTML of each article from the ZIM file, parsing it to extract clean text, and then chunking it into smaller, semantically coherent pieces (e.g., by section heading). These chunks would then be stored in the app's primary SQLite database, along with pre-computed vector embeddings for semantic search. This hybrid approach combines the efficient packaging and compression of the ZIM format with the advanced search and data management capabilities of SQLite, resulting in a highly performant and comprehensive offline medical reference system.

4.3. Content Packaging and Preprocessing

4.3.1. Extracting and Chunking Content from ZIM Files

The ZIM file format, while excellent for packaging and compressing large volumes of content, is not directly optimized for the kind of granular, semantic search required by the emergency app. The articles within a ZIM file are stored as complete HTML documents, which can be quite large and contain a mix of content, including navigation, infoboxes, and references. To make this content more useful for a mobile application, a preprocessing step is required to extract the raw text and break it down into smaller, more manageable chunks. This process begins with using a library like `python-libzim` to iterate through the articles in the ZIM file and extract their HTML content. Once the HTML is obtained, it needs to be parsed and cleaned. Tools like `lxml` or `BeautifulSoup` are ideal for this task, as they can be used to strip out all the non-essential HTML tags, scripts, and styles, leaving only the clean, readable text of the article's body. This is a crucial step, as it significantly reduces the size of the data and makes it easier to work with in subsequent stages. The next step is to chunk the cleaned text into smaller, semantically coherent units. A simple approach would be to split the text into fixed-size blocks of a certain number of characters or tokens. However, a more sophisticated method is to use the structure of the article itself. By identifying section headings (e.g., `<h2>` , `<h3>` tags), the text can be split at these natural boundaries, preserving the logical flow of the article. This structure-aware

chunking is particularly important for medical articles, where different sections might cover symptoms, causes, diagnosis, and treatment. By preserving this structure, the app can provide more relevant and contextually appropriate information to the user. For example, if a user searches for "symptoms of a heart attack," the app can return the specific chunk of text from the "Symptoms" section of the heart attack article, rather than a large, undifferentiated block of text from the entire article.

The technical implementation of the extraction and chunking pipeline is a critical part of the app's build process and should be handled on a server or development machine, not on the mobile device itself. The pipeline would start with a script that uses `python-libzim` to open the selected medical ZIM file and iterate through its directory entries. For each article, the script would retrieve the HTML content and pass it to a parsing function. This function, using a library like `BeautifulSoup`, would identify and remove all non-content elements, such as the page header, footer, navigation menus, and any sidebars or infoboxes. It would then extract the main body text and identify all the section headings. The text would then be split into chunks based on these headings. A good practice is to include the heading text as metadata for each chunk, which can be useful for both search and display purposes. The size of the chunks is also an important consideration. For the purpose of generating embeddings for semantic search, a chunk size of around 512 tokens is a common choice. To ensure that no important information is lost at the boundaries between chunks, a small overlap of 50–128 tokens (a 25% overlap ratio) is recommended. This means that the last 50–128 tokens of one chunk will also be the first 50–128 tokens of the next chunk. The final output of this pipeline would be a structured dataset, likely in JSON or CSV format, where each entry represents a chunk of text and includes the article title, the section heading, the chunk text itself, and any other relevant metadata. This structured data is then ready to be ingested into the app's SQLite database and used for generating embeddings.

4.3.2. Generating Embeddings for Semantic Search

To enable a more intelligent and user-friendly search experience beyond simple keyword matching, the app will incorporate semantic search. This allows the app to understand the intent and contextual meaning of a user's query, rather than just matching exact words. For example, a search for "my chest hurts" should return results related to heart attacks, even if the word "heart" is not in the query. The foundation of semantic search is the use of vector embeddings, which are numerical representations of text that capture its semantic meaning. These embeddings are generated by a

machine learning model, and for the emergency app, a lightweight and efficient model is required to stay within the resource constraints. The `all-MiniLM-L6-v2` model is an excellent choice for this purpose. It is a small, fast model that produces high-quality 384-dimensional embeddings, and it has a size of only around 22MB, making it suitable for on-device use. However, to avoid the computational overhead and battery drain of generating embeddings in real-time on the user's device, the recommended approach is to pre-compute the embeddings for all the text chunks at build time. This means that the embedding generation process is run once on a server as part of the content preprocessing pipeline. The resulting vectors are then stored directly in the app's SQLite database alongside the corresponding text chunks. This allows the search process to be extremely fast and efficient. When a user enters a query, the app only needs to generate a single embedding for the query text. It can then use a vector search library, such as `sqlite-vec`, to quickly find the pre-computed embeddings in the database that are most similar to the query embedding. This similarity is typically measured using a distance metric like cosine similarity or Euclidean distance. By pre-computing the embeddings, the app can provide a powerful semantic search experience with minimal impact on the device's performance or battery life.

The implementation of the embedding generation pipeline involves several steps. First, the `all-MiniLM-L6-v2` model needs to be downloaded and integrated into the build-time processing script. This script will take the structured dataset of text chunks produced by the extraction and chunking pipeline as input. For each text chunk, the script will pass the text through the model to generate a 384-dimensional vector of floating-point numbers. This vector is the embedding. The script will then store this embedding in the SQLite database, associating it with the corresponding text chunk. The `sqlite-vec` extension is particularly useful here, as it provides a way to store and search these vectors directly within SQLite, without the need for a separate vector database. The vectors can be stored as BLOBs in a dedicated column of the table containing the text chunks. The `sqlite-vec` extension provides functions to perform efficient vector similarity searches. For example, a query could be written to find the 10 text chunks whose embeddings have the smallest cosine distance to the query embedding. The storage overhead for these embeddings is relatively modest. Each 384-dimensional vector, when stored as a BLOB of 32-bit floats, consumes 1,536 bytes (384 * 4). For a database of 10,000 text chunks, the total storage required for the embeddings would be approximately 15MB, a small fraction of the app's total 500MB budget. This investment in storage provides a significant return in terms of

search quality and user experience, allowing users to find the information they need quickly and intuitively, even if they don't know the exact medical terminology.

4.3.3. Building a Pre-Processed SQLite Knowledge Base

The culmination of the content strategy is the creation of a single, self-contained SQLite database that serves as the app's offline knowledge base. This database is the central repository for all the pre-processed and optimized content, including the chunked text from ZIM files, the pre-computed vector embeddings for semantic search, and the full-text search index. The process of building this database is a critical part of the app's build pipeline and is performed on a server or development machine. The pipeline begins by taking the structured output from the content extraction and chunking process. This data, which includes the article title, section heading, and the cleaned, chunked text, is then inserted into a table in the SQLite database. A schema for this table might include columns for `id`, `article_title`, `section_heading`, `chunk_text`, and `embedding`. The next step is to generate the vector embeddings for each text chunk using the `all-MiniLM-L6-v2` model, as described previously. These embeddings are then stored in the `embedding` column of the table. To enable fast full-text search, an FTS5 virtual table is created. The FTS5 extension is highly efficient and can be configured to create a contentless index, which minimizes storage overhead by not storing a second copy of the text. Instead, it stores a reference to the original row in the main table. The FTS5 index is configured to use the Porter stemming algorithm for English content, which improves search recall by matching different forms of a word (e.g., "run" and "running"). Finally, the entire database file is compressed using a tool like Zstandard with a trained dictionary. This can achieve very high compression ratios, especially on the homogeneous text of medical articles, further reducing the size of the data that needs to be bundled with the app. The final, pre-processed SQLite database is a highly optimized, all-in-one package that contains all the medical reference content, ready for fast, offline access. This approach ensures that the app can provide a rich, responsive, and intelligent search experience without requiring any complex data processing or network access on the user's device.

The architecture of the pre-processed SQLite database is designed for maximum performance and efficiency in a read-heavy, offline environment. The primary table, which stores the text chunks, is structured to allow for quick retrieval of content based on an ID. The `sqlite-vec` extension is used to create a virtual table for managing the vector embeddings. This allows for efficient K-Nearest Neighbor (KNN) searches to

find the most semantically similar text chunks to a user's query. The integration between the main table and the vector table is seamless, as the vector search can return the IDs of the matching rows, which can then be used to fetch the corresponding text from the main table. The FTS5 full-text search index is another key component of the database. It is created as a separate virtual table that indexes the `chunk_text` column of the main table. This allows the app to perform fast, complex text searches, including phrase searches and prefix searches. The FTS5 index is configured with `detail=none` to minimize its storage footprint, as the full text is already available in the main table. The combination of these three components—the main content table, the vector search table, and the full-text search index—provides a powerful and flexible search capability. A user query can be processed by first generating an embedding for the query and performing a semantic search to find the most relevant chunks. This can be combined with a traditional full-text search to ensure that all relevant results are found. The results from both searches can then be merged and ranked to provide a final, comprehensive list of results. This multi-modal search approach ensures that users can find the information they need, whether they are using natural language, specific keywords, or a combination of both. The entire database is then compressed and bundled with the app, providing a complete, self-contained knowledge base that is ready for immediate use, even in the most challenging offline scenarios.

5. Mapping and Geolocation

5.1. Offline Mapping Strategy

5.1.1. Leveraging OpenStreetMap Data

The foundation of any offline mapping solution is the underlying geographic data. For an emergency preparedness app that must operate entirely without an internet connection, the choice of data source is paramount. **OpenStreetMap (OSM)** emerges as the ideal candidate due to its open license, global coverage, and rich feature set . Unlike proprietary mapping services that restrict redistribution or require constant API calls, OSM data is freely available under the Open Database Licence (ODbL), which allows for the creation and distribution of derivative works, including fully offline map packages. This legal freedom is a non-negotiable prerequisite for building a truly air-gapped application. The OSM dataset is a collaborative, community-driven project that contains a vast amount of geographic information, including roads, buildings, points of interest (POIs), and administrative boundaries. For the UK-focused emergency app, this data can provide detailed street-level maps, the locations of hospitals, police stations,

and other critical infrastructure, all of which are essential for effective emergency response.

The raw OSM data, typically in the `.osm.pbf` format, is a large and complex XML-like dataset that is not directly usable by mobile map rendering libraries. Therefore, it must be processed and converted into a more efficient format for mobile use. This is where projects like **OpenMapTiles** come into play. OpenMapTiles provides a set of open-source tools and schemas for converting raw OSM data into **vector tiles**. These tiles are small, pre-rendered chunks of map data (typically in the `.pbf` format) that contain all the necessary information to draw a specific area of the map at a specific zoom level. By packaging the map as a collection of these vector tiles, the app can render the map locally on the device with high performance and minimal storage overhead. The use of vector tiles, as opposed to raster tiles (static images), also allows for dynamic styling, where the appearance of the map (colors, fonts, line weights) can be changed on the fly without needing to download new tile data. This flexibility is a significant advantage for an emergency app, where different visual styles might be used to highlight different types of information, such as flood zones or evacuation routes.

5.1.2. Vector Tiles with OpenMapTiles and MapLibre

Once the decision to use OpenStreetMap data has been made, the next step is to select the technology stack for rendering the vector tiles on the mobile device. The combination of **OpenMapTiles** for data processing and **MapLibre** for rendering provides a powerful, open-source, and fully offline-capable solution. OpenMapTiles is not just a data format; it is a complete ecosystem for generating vector tiles from OSM data. It provides a well-defined schema that dictates how geographic features (like roads, buildings, and POIs) are encoded into the tile format. This standardization ensures compatibility with a wide range of rendering libraries. The process involves using OpenMapTiles tools to extract the relevant geographic region (e.g., the United Kingdom) from the full OSM planet file and generate a set of vector tiles for the desired zoom levels (e.g., zoom levels 0-14 for a good balance of detail and file size). The resulting tileset, typically packaged as an MBTiles file or a directory of `.pbf` files, forms the core of the app's offline map data.

For rendering these tiles on the device, **MapLibre** is the recommended choice. MapLibre is an open-source, community-driven fork of the popular Mapbox GL Native rendering engine. It is designed to be a drop-in replacement for Mapbox's proprietary SDK, offering the same high-performance, hardware-accelerated rendering of vector

tiles but without the licensing restrictions or API key requirements. MapLibre can consume the vector tiles generated by OpenMapTiles and render them into a fully interactive map view. It supports all the key features of a modern mapping library, including smooth panning and zooming, tilt and rotation, and the application of custom styles. A style is a JSON file that defines the visual appearance of the map, specifying how each type of geographic feature (e.g., motorways, parks, hospitals) should be drawn. By bundling a custom style with the app, the developers can create a map that is optimized for the emergency use case, with clear visual hierarchies and prominent highlighting of critical locations. The combination of OpenMapTiles for data and MapLibre for rendering provides a complete, end-to-end, offline mapping solution that is both powerful and compliant with the open-source ethos of the project .

5.1.3. Comparison with Proprietary Solutions (Mapbox)

While the open-source combination of OpenMapTiles and MapLibre is the recommended path, it is instructive to compare it with proprietary solutions, most notably **Mapbox**. Mapbox offers a highly polished and feature-rich mapping SDK that is widely used in the industry. It also provides robust support for offline maps through its **Offline Manager API** . This API allows developers to define specific geographic regions and download the corresponding map tiles, styles, and other resources for offline use. The process is well-documented and relatively straightforward, involving the creation of "style packs" and "tile regions" that are managed by the SDK . For developers already familiar with the Mapbox ecosystem, this can be an attractive option.

However, the use of Mapbox comes with significant trade-offs that make it less suitable for the specific goals of this emergency preparedness app. The most significant drawback is the **licensing and cost**. Mapbox's terms of service do not allow for the redistribution of pre-packaged offline data. This means the app cannot be shipped with the map data already included in the bundle; users must download it themselves after installation. Furthermore, while Mapbox offers a generous free tier, usage beyond that is billed based on API requests (e.g., for tile downloads). For an app that may be downloaded by thousands of users, this could lead to unpredictable and potentially high costs. Another issue is the requirement for a **Mapbox access token**. While it is technically possible to use the Mapbox GL SDK with self-hosted tiles without a token, the official offline download mechanisms require one, adding a layer of complexity and dependency on a third-party service . In contrast, the OpenMapTiles and MapLibre solution is entirely self-contained and free of licensing fees, aligning perfectly with the project's goal of creating a truly independent, air-gapped application.

While Mapbox offers a more integrated and "batteries-included" experience, the freedom, cost-effectiveness, and offline-first nature of the open-source alternative make it the superior choice for this specific use case.

5.2. Implementation on iOS

5.2.1. Integrating MapLibre for Local Vector Tile Rendering

The implementation of the offline mapping feature on iOS will leverage the **MapLibre Mobile SDK**, which is available for easy integration via the **Swift Package Manager** . This modern dependency management system simplifies the process of adding the library to the Xcode project and ensures that the app is always using a consistent and versioned build of the SDK. The core of the implementation involves creating a `MLNMapView` instance, which is the primary view component responsible for rendering the map. This view is then configured to use a local data source for its tiles, styles, and other assets, completely bypassing any need for a network connection. The key to this offline configuration lies in the use of the **`asset://` URI scheme**. This scheme tells the MapLibre renderer to look for the specified resources within the app's own bundle, rather than fetching them from a remote server.

The configuration is typically done through a style JSON file. This file, which is also bundled with the app, contains all the instructions for the renderer, including the source of the vector tiles. An example configuration within the style JSON would look like this :

JSON 复制

```
"sources": {
    "openmaptiles": {
        "type": "vector",
        "tiles": [
            "asset://data/tiles/{z}/{x}/{y}.pbf"
        ]
    }
}
```

This configuration points the map renderer to a directory named `tiles` within the app's `data` bundle, where the pre-generated `.pbf` vector tiles are stored. The `{z}` , `{x}` , and `{y}` placeholders are standard tile coordinates that allow the renderer to request the specific tiles it needs to display the current map view. Similarly, the style file will use the `asset://` scheme to reference the local sprite files (for

icons) and glyph files (for fonts), ensuring that every component required to render a complete and styled map is available locally. This approach results in a 100% offline mapping experience, with the map loading and rendering instantly, as it does not need to wait for any network requests. The integration is straightforward and follows standard iOS development practices, making it a reliable and maintainable solution.

5.2.2. Packaging Vector Data within the App Bundle

A key architectural decision for the offline map implementation is how to package the vector tile data within the iOS application's bundle. The goal is to include the necessary geographic data for the target region (the UK) in a format that is both efficient in terms of storage and performant in terms of access. The recommended approach, as demonstrated in the `openmaptiles-ios-demo` project, is to include the pre-generated vector tiles as a set of individual `.pbf` files organized in a directory structure that mirrors the standard XYZ tiling scheme . This directory, containing all the tiles for the desired zoom levels, is then added to the Xcode project as a folder reference. When the app is built, this folder and its contents are copied directly into the app's main bundle.

This method of packaging the data as individual files, rather than a single large archive like an MBTiles file, offers a significant performance advantage. When the MapLibre renderer needs to display a specific tile, it can directly access the corresponding `.pbf` file from the bundle using a simple file path constructed from the tile coordinates. This avoids the overhead of having to open a large database file and query it for the required tile data, resulting in faster map loading and smoother panning and zooming. The total size of the vector tile data for the UK at zoom levels 0–14 is estimated to be in the range of **200–300MB**, which fits comfortably within the overall 500MB budget for the application . This size provides a good level of detail, showing major roads, towns, and points of interest, which is sufficient for the app's emergency use case. By pre-packaging the data in this way, the app guarantees that the map is fully functional from the moment of installation, with no additional downloads required, fulfilling the core requirement of a truly offline-first experience.

5.3. Cross-Platform Considerations

5.3.1. React Native Integration with react-native-maps

For a project with potential cross-platform ambitions, it is important to consider how the offline mapping solution could be implemented within a framework like **React**

Native. The most popular and widely used mapping library in the React Native ecosystem is **react-native-maps** . This library provides a unified JavaScript API for integrating maps into a React Native application, with support for both Google Maps and Apple Maps as the underlying rendering engine on iOS and Android. While react-native-maps is primarily designed for online use, it does offer a component called **LocalTile** that can be used to implement offline mapping functionality . This component allows the developer to specify a pathTemplate that points to a local directory of tile files on the device's file system.

The LocalTile component works by overlaying the local tiles onto the base map. To create a fully offline experience, the base map's type should be set to "none" to hide the default online tiles. The pathTemplate uses the standard {z} , {x} , and {y} placeholders to locate the correct tile file for the current map view. For example, a path template might look like file:///storage/emulated/0/maps/{z}/{x}/{y}.png on Android or ${NSDocumentDirectory}/maps/{z}/{x}/{y}.png on iOS . This approach allows the app to render a map using pre-packaged raster tiles (.png or .jpg files). However, this method has a significant drawback: raster tiles are much less efficient in terms of storage than vector tiles. As noted in a discussion on the topic, a raster tile set for a single region at a good zoom level can easily consume **3GB of space** or more, which is far too large for the 500MB budget of this project . While react-native-maps is a viable option for simple offline maps, its reliance on raster tiles for the LocalTile component makes it unsuitable for this data-constrained application. A more advanced solution, such as a React Native wrapper around MapLibre, would be required to leverage the efficiency of vector tiles in a cross-platform context.

5.3.2. Android Implementation with Play Asset Delivery

On the Android platform, the mechanism for delivering large amounts of data alongside the application binary is **Play Asset Delivery (PAD)** . This is the modern, recommended replacement for the deprecated APK Expansion Files. Play Asset Delivery is deeply integrated with the Google Play Store and allows developers to ship assets up to **1GB in size** (with a total limit of 4GB) in a more flexible and efficient manner. PAD offers three distinct delivery modes, which can be chosen based on the criticality of the data . The **install-time** delivery mode is the most relevant for the core emergency content of this app. Assets configured for install-time delivery are automatically downloaded alongside the APK when the user installs the app from the Play Store. This ensures that the essential data, such as the SQLite databases and conversation trees, is present on

the device from the very first launch, mirroring the functionality of the iOS Background Assets framework.

For the large map dataset, the **fast-follow** delivery mode could be used. Assets configured for fast-follow are automatically downloaded immediately after the app is installed, but they do not block the initial launch. This would allow the user to start using the core features of the app right away, while the map data is downloaded in the background. Once the download is complete, the app would be notified and could then enable the full mapping functionality. The third mode, **on-demand**, allows the app to request and download asset packs at runtime. This could be used for downloading map data for additional regions that the user might travel to. For apps that are distributed outside of the Google Play Store (i.e., sideloaded), Play Asset Delivery is not available. In this scenario, a custom download solution would need to be implemented, likely using the **WorkManager** API to manage large, background downloads with reliability and resiliency. This custom solution would need to handle its own manifesting, downloading, and delta-updating logic, making it a more complex undertaking than using the Play Store's built-in mechanisms.

6. Reference Implementations and Proven Patterns

6.1. Offline Knowledge Apps

6.1.1. Kiwix: A Model for Offline Content Distribution

The **Kiwix** project stands as a paramount reference for building a robust offline knowledge base, offering a proven model for content packaging, distribution, and access that is directly applicable to the emergency preparedness app . Kiwix's core innovation is its use of the **ZIM file format**, a highly compressed, open-standard container designed specifically for storing web content for offline consumption. This format allows for the bundling of massive datasets, such as the entirety of Wikipedia, into a single, manageable file that can be easily downloaded and distributed. The project's primary goal is to make knowledge accessible to users with limited or no internet connectivity, a mission that aligns perfectly with the "airgapped" requirement of the emergency app. The Kiwix Library offers a vast catalog of pre-packaged ZIM files covering a wide range of topics, from general encyclopedias to specialized resources like medical texts and educational materials, providing a rich source of content that can be selectively integrated into the app .

From an architectural perspective, the Kiwix model provides several key insights. Firstly, it demonstrates the power of a **modular content package architecture**. Users can download only the content packages (ZIM files) they need, which is a crucial pattern for managing the app's 500MB budget. The Kiwix iOS app, for example, allows users to browse a catalog of available content and download specific packages for offline use . This pattern can be directly translated to the emergency app, where users might download a "UK Emergency Law" package, a "Medical First Aid" package, and a "UK Maps" package. Secondly, Kiwix's approach to handling large, compressed data on mobile devices is instructive. While the project provides tools to read ZIM files directly, the recommended strategy for the emergency app is to **preprocess the ZIM files at build time**, extracting the relevant articles and importing them into a more query-friendly SQLite database. This hybrid approach combines the efficient distribution of ZIM files with the high-performance search capabilities of a dedicated database, creating a powerful and user-friendly offline experience.

6.1.2. Organic Maps: Patterns for Offline Map Data

Organic Maps is an open-source, offline-first mapping application that serves as an excellent reference for the design and implementation of the app's mapping component. The app, which has over 12,000 stars on GitHub, is built on a foundation of OpenStreetMap data and provides a rich set of features, including offline geocoding, POI search, and turn-by-turn navigation, all without requiring an internet connection. A key architectural pattern demonstrated by Organic Maps is its use of a **binary map file format (`.mwm`)** . This format is a highly optimized, compressed container for map data that is derived from OpenStreetMap. By using a custom binary format, Organic Maps can achieve a very small file size for its map data, which is critical for an offline-first application. The app also implements a **region-based download system**, where users can download maps for specific countries or regions, allowing them to manage their device's storage effectively.

Another important pattern from Organic Maps is its approach to **incremental updates**. The app uses a delta update system to minimize the amount of data that needs to be downloaded when a map is updated. This is a crucial feature for an app that may be used in areas with limited or expensive bandwidth. The user interface of Organic Maps also provides valuable insights. The app clearly indicates which map regions have been downloaded and shows the storage usage for each region, giving the user full control over their offline data. These patterns—a custom binary data format, region-based downloads, incremental updates, and a transparent user interface for managing offline

content—are all directly applicable to the design of the emergency preparedness app. By studying and adapting these proven patterns, the development team can create a mapping component that is both powerful and user-friendly, providing a seamless offline experience that is essential for an emergency preparedness tool.

6.2. Emergency and First Aid Apps

6.2.1. British Red Cross First Aid App: A Size Benchmark

In the landscape of offline-capable emergency applications, the British Red Cross First Aid app serves as a crucial reference point, particularly for understanding the potential size and scope of a feature-rich, content-heavy application. While the primary development target is iOS, analysis of the Android version provides a valuable and highly relevant benchmark. According to data from an APK download repository, the Android application package for the "First aid by British Red Cross" app has a file size of **163 MB** . This figure is significant as it represents a real-world example of an app that bundles a substantial amount of first aid content—including guides, videos, and quizzes—for offline use. The app's description explicitly states that it works without an internet connection, making it a direct analogue for the proposed project . The 163 MB size, which covers over 20 first aid skills, provides a strong indication that a comprehensive offline first aid resource can be delivered within a reasonable footprint. This benchmark is particularly useful for validating the initial size budget allocation for the proposed app, which aims to stay within a 500 MB total constraint. The British Red Cross app's size suggests that a significant portion of the budget can be allocated to the core first aid and emergency content, while still leaving ample room for other essential components such as the on-device machine learning model, the mapping data, and the application binary itself.

Further analysis of the British Red Cross's app ecosystem reveals a more granular view of content packaging. The organization also offers a more specialized "Baby and Child First Aid" app, which has a significantly smaller footprint. On the iOS App Store, this specific app is listed with a size of **51.54 MB** or **52.7 MB** . This smaller size is logical, as the content is focused on a narrower set of scenarios. The existence of both a general-purpose app (163 MB on Android) and a specialized, smaller app (approx. 52 MB on iOS) demonstrates a strategic approach to content segmentation. This pattern could be emulated in the proposed project, perhaps by offering modular content downloads or separate, targeted versions of the app. The size difference also highlights the impact of media-rich content, such as videos and high-resolution illustrations, on the overall app size. The general app likely contains more and longer videos to cover a

wider range of scenarios, contributing to its larger footprint. This insight is critical for the content strategy of the new app, as it underscores the need to carefully balance the richness of the content with the strict size constraints. The decision to include video, for example, must be weighed against the impact on the total app size and the potential for using more compact, text-and-image-based guides. The British Red Cross apps, therefore, provide not just a single data point, but a valuable case study in how to structure and size an offline emergency preparedness application.

6.2.2. St John Ambulance First Responder App: Feature Analysis

The St John Ambulance "First Responder" app, available on the Google Play Store, provides another valuable case study in the design of a comprehensive emergency response application. While the app's internal architecture and size are not detailed in the store listing, its feature set offers a clear blueprint for the types of functionalities that are considered essential in a modern emergency app . The app is explicitly designed to be a tool for both self-help and community support, and its features can be categorized into several key areas. First, it includes a robust **First Aid Guide** section, which provides searchable, step-by-step instructions with illustrations for a range of emergencies. This core feature is enhanced with a CPR timer and the ability for users to add guides to a favorites list for quick access. Second, the app has a **First Responder** registration system, which allows trained first aiders to sign up to receive alerts about nearby emergencies within a 500-meter radius, enabling them to provide assistance before professional help arrives. This community-oriented feature adds a layer of social responsibility and utility beyond simple reference. Third, it integrates directly with emergency services, providing a one-touch button to call Triple Zero (000) with automatic location detection. This integration is a critical safety feature that ensures users can escalate to professional help instantly.

The app's feature set extends beyond immediate first aid and emergency calling to include preparedness and resource location tools. A key component is the **Defibrillators (AED) Map**, which helps users find the nearest automated external defibrillator in an emergency. This map is crowdsourced, allowing users to upload the locations of new defibrillators to the network, thereby improving the resource for the entire community. The app also provides information on the nearest **Emergency Departments**, including live waiting times, contact information, and directions. This feature, which likely requires a network connection for real-time data, demonstrates a hybrid approach where some dynamic information is layered on top of a core offline foundation. Additionally, the app includes information on St John Urgent Care centers and General

Practices, as well as a booking feature for patient transport services. The combination of offline first aid guides, community responder features, direct emergency service integration, and resource mapping provides a holistic view of what a top-tier emergency app can offer. For the development of the proposed app, the St John Ambulance app serves as an excellent model for feature prioritization and user experience design, particularly in its integration of offline reference material with online, real-time services.

6.2.3. IFRC First Aid App: Target UX Patterns

The International Federation of Red Cross and Red Crescent Societies (IFRC) First Aid app, which forms the basis for many national Red Cross first aid apps around the world, represents a global standard for emergency preparedness applications . While the app itself is not open source, its documented features and the experiences of national societies that have adopted it provide a clear picture of the target user experience (UX) patterns that should be emulated. A central tenet of the IFRC app's design is the provision of **preloaded content**, ensuring that users have instant access to critical, life-saving information even in the absence of mobile connectivity . This offline-first approach is the foundational principle of the proposed app. The content itself is based on the latest international first aid and CPR guidelines, and it is presented in a simple, step-by-step format that is easy to follow under pressure. The app also places a strong emphasis on **accessibility**, using person-first, non-gendered language and focusing on graphics and artwork to depict skills and techniques, making the information understandable to a wider audience, including those with literacy challenges or disabilities .

Another key UX pattern demonstrated by the IFRC app and its derivatives is the seamless integration of learning and reference modes. The app is not just a static reference tool; it is designed to help users **maintain and test their skills** through interactive quizzes and training modules . This dual-mode design, which separates the "learn" function from the "emergency reference" function, is a critical pattern to adopt. In a calm, non-emergency setting, users can engage with quizzes and tutorials to build their confidence and knowledge. In a real emergency, they can instantly access the step-by-step guidance they need without having to navigate through educational content. The app also includes practical features such as an "I'm Safe" button, which allows users to send a pre-written message to their loved ones to let them know they are out of harm's way, and direct integration with local emergency numbers (e.g., 911 in the US, 999 in the UK) . The Hong Kong Red Cross version of the app, for example, is

fully integrated with the 999 hotline, allowing for in-app dialing . These patterns—offline-first content, accessible design, dual-mode learning/reference functionality, and integrated communication tools—collectively define the gold standard for emergency app UX and should serve as the primary guide for the design of the new application.

6.3. Chat and Messaging SDKs

6.3.1. Stream Chat SDK: Offline-First Architecture Patterns

While the primary UI for the app will be built using MessageKit on iOS and a custom RecyclerView on Android, the **Stream Chat SDK** provides an excellent reference for designing a robust, offline-first architecture for messaging and conversational interfaces. The Stream SDK is a commercial product, but its documentation and open-source components offer valuable insights into how to build a scalable and reliable chat system. A key feature of the Stream SDK is its comprehensive **offline support**. The SDK uses a combination of **SQLite and Room** for persistence on Android, and a similar local database solution on iOS, to store messages, channels, and user data locally on the device. This ensures that the chat interface is always functional, even when there is no network connectivity.

The SDK's architecture is built around several key patterns that are relevant to the emergency app. First, it implements **optimistic UI updates**. When a user sends a message, the UI is updated immediately to show the message as sent, even before it has been confirmed by the server. This provides a more responsive and satisfying user experience. Second, it uses an **automatic retry queue** for failed messages. If a message cannot be sent due to a network error, it is placed in a queue and the SDK will automatically attempt to resend it when the connection is restored. This ensures that no messages are lost due to temporary network issues. Finally, the SDK uses reactive programming patterns, such as **StateFlow and LiveData** on Android, to provide a reactive stream of data to the UI. This means that the UI is automatically updated whenever the underlying data in the local database changes, simplifying the process of keeping the interface in sync with the data. The `StreamOfflinePluginFactory` pattern provides a clean abstraction for this offline-first sync logic, separating the concerns of data persistence and network synchronization from the UI layer. While the full Stream SDK may be too heavyweight for the emergency app, these architectural patterns provide a valuable blueprint for building a reliable and user-friendly conversational interface.