Технический анализ: Пять вероятных причин сбоя механизма повторной загрузки файлов в производственной среде (P1)

Введение

Цель отчета

Настоящий отчет представляет собой детальный технический анализ проблемы P1 — повторяющегося сбоя механизма повторной загрузки файлов в мобильном приложении. Цель анализа — выявить пять наиболее вероятных коренных причин, оценить их вероятность и предоставить обоснования, подкрепленные фактическими данными, для направления усилий по устранению неисправности.

Аналитическая основа

Анализ строится на двух ключевых концепциях, которые объясняют расхождение в поведении системы между контролируемой лабораторной средой и реальными условиями эксплуатации («в поле»):

1. **Каскад волатильности:** Наблюдаемый сбой — это не единичная ошибка, а каскадный отказ, возникающий в результате взаимодействия нескольких нестабильных компонентов: эфемерных URI файлов, ненадежных сетевых соединений, временного состояния приложения и ограниченных по времени жизни токенов безопасности. Каждый из этих элементов в отдельности может быть управляемым, но их совокупное воздействие в условиях

эксплуатации приводит к системному сбою.
2. **Дихотомия «лаборатория vs. поле»:** Суть проблемы заключается в условиях, уникальных для полевой эксплуатации: плохое качество сетевого соединения, непредсказуемые события жизненного цикла приложения (переход в фоновый режим, принудительное завершение операционной системой) и значительные временные задержки между первоначальной попыткой загрузки и ее повтором. Эти условия не воспроизводятся в достаточной мере в контролируемых лабораторных тестах.

Диагностический признак: «undefined values»

Сообщение в логах об «undefined values» (неопределенных значениях) рассматривается не как прямая причина, а как конечный симптом. Данный анализ продемонстрирует, как каждая из пяти выявленных коренных причин может приводить к появлению именно этого сообщения об ошибке, раскрывая его как диагностический ключ, а не как указание на буквальную ошибку в коде.

Причина №1: Недействительность эфемерных URI файлов после выбора

- **Вероятность:** Очень высокая (90%)
- **Основная гипотеза:** Механизм повторной загрузки пытается использовать URI файла, который более не является действительным. Библиотека expo-image-picker по умолчанию возвращает URI, указывающий на временный файл в кэше приложения. Нет гарантии, что этот URI будет существовать постоянно, особенно после перезапуска приложения или если операционная система очистит кэш.

Детальный анализ

Природа временных URI

Ключевым моментом является понимание того, что библиотеки для выбора изображений, такие как expo-image-picker, предоставляют URI не на исходное изображение в галерее устройства, а на его временную копию, созданную в изолированном пространстве приложения.[1] Эта копия создается для того, чтобы приложение могло безопасно работать с файлом, не запрашивая широких разрешений на доступ ко всему хранилищу пользователя. Однако жизненный цикл этой копии и ее URI строго ограничен. Документация и опыт сообщества показывают, что нельзя полагаться на постоянство этих URI. В частности, на Android временный путь (

response.uri) может быть удален при закрытии приложения, а на iOS идентификатор приложения (GUID), являющийся частью пути к файлу, может измениться при перезапуске, делая сохраненный абсолютный URI недействительным.[1]

Специфика схемы content:// в Android

На современных версиях Android проблема усугубляется использованием схемы URI content://. В отличие от прямого пути к файлу (file://), content:// URI является указателем на Content Provider — системный механизм для безопасного обмена данными между приложениями.[3] Доступ к такому URI предоставляется приложению на временной основе, как правило, на время жизни текущей «активности» (Activity). Как только пользователь закрывает приложение или система принудительно его завершает, это временное разрешение отзывается. При последующем запуске приложения и попытке использовать сохраненный

content:// URI для доступа к файлу, операционная система откажет в доступе, поскольку разрешение более недействительно.

Паттерн «Копирование в постоянное хранилище»

Правильным архитектурным решением этой проблемы является немедленное копирование файла из временного местоположения, предоставленного expo-image-picker, в постоянный каталог в «песочнице» приложения. Expo предоставляет два основных каталога: FileSystem.cacheDirectory для временных файлов, которые система может удалить при нехватке памяти, и FileSystem.documentDirectory для постоянных файлов, которые удаляются только самим приложением.[4] Задача состоит в том, чтобы сразу после получения временного URI использовать функцию, такую как

FileSystem.copyAsync() или file.move(), для перемещения файла в FileSystem.documentDirectory.[4] Именно URI этой

постоянной копии должен сохраняться в очереди на повторную отправку. Даже если используется опция copyToCacheDirectory в пикере, файл все равно оказывается во временном кэше, который может быть очищен системой, что делает этот подход ненадежным для отложенных задач.[7]

Как это приводит к «undefined values»

Цепочка событий, ведущая к ошибке, выглядит следующим образом. Логика повторной отправки извлекает из очереди устаревший, недействительный URI. Попытка прочитать файл по этому URI с помощью файловой системы завершается неудачей (например, выбрасывается исключение или возвращается null). Если блок catch или последующая логика обработки ошибок не реализованы достаточно надежно, переменная, которая должна была содержать двоичные данные файла или объект файла, получает значение undefined. Затем эта неопределенная переменная передается в функцию загрузки (например, в тело fetch запроса), что и вызывает зафиксированную в логах ошибку «undefined values».

Это фундаментальная проблема целостности данных на «первой миле». Если ссылка на исходные данные является хрупкой, никакая, даже самая изощренная, логика повторных попыток или управления очередью не сможет исправить ситуацию. Сбой происходит в самом начале офлайн-процесса. Недавнее исправление, которое сработало в лаборатории, вероятно, не включало в себя

полный цикл перезапуска приложения. В рамках одного сеанса работы приложения временный URI оставался действительным, создавая ложное впечатление, что проблема решена. Однако это исправление не учитывает полный жизненный цикл приложения, который регулярно происходит в полевых условиях, и поэтому является неполным.

Причина №2: Истечение срока действия SAS-токена хранилища Azure Blob при отложенной повторной попытке

- **Вероятность:** Высокая (80%)
- **Основная гипотеза:** Механизм повторной загрузки использует тот же токен разделенного доступа (Shared Access Signature, SAS), который был сгенерирован для первоначальной неудачной попытки. Из-за значительной временной задержки, характерной для полевых операций, к моменту выполнения повторной попытки этот токен истекает, что приводит к ошибке аутентификации на стороне Azure.

Детальный анализ

Паттерн «Valet Key» и жизненный цикл SAS-токена

Для организации загрузки файлов на стороне клиента стандартным и безопасным методом является паттерн «Valet Key» (Ключ-камердинер). Суть его в том, что мобильное приложение не хранит мощные ключи доступа к хранилищу Azure. Вместо этого оно обращается к собственному бэкенду, который генерирует и возвращает краткосрочный, ограниченный по правам доступа SAS-токен специально для данной операции.[8] Этот токен делегирует приложению временное право на выполнение конкретных действий (например, запись одного конкретного файла).[9]

Истечение срока действия — это функция безопасности

Критически важно понимать, что ограниченный срок действия SAS-токена — это не ошибка, а фундаментальная функция безопасности. Установка короткого времени жизни (например, от 5 до 30 минут) является лучшей практикой, поскольку это минимизирует риск злоупотребления в случае компрометации токена.[10] Такого времени жизни вполне достаточно для немедленной загрузки в условиях стабильного соединения. Однако оно абсолютно неадекватно для сценария повторной попытки, которая может произойти через несколько часов, после того как техник потерял связь, переместился и снова оказался в зоне покрытия сети.

Как это приводит к «undefined values»

Попытка загрузить файл с использованием просроченного SAS-токена приведет к тому, что Azure Blob Storage вернет ошибку аутентификации, как правило, со статусом HTTP 403 Forbidden.[11] Если сетевой слой приложения или логика обработки ошибок не настроены на специфическую обработку этого кода состояния, ошибка может быть воспринята как общий сбой сети. Код, ожидающий в ответе либо данные об успешной загрузке, либо определенную структуру ошибки, может вместо этого получить

null или undefined. Эта неопределенность затем распространяется по цепочке вызовов, пока не приведет к фатальной ошибке «undefined values» при попытке доступа к свойствам несуществующего объекта ответа. Например, код response.data.url упадет, если response будет undefined.

Правильный паттерн повторной попытки

Архитектурно верный подход заключается в том, что логика повторной попытки не должна повторно использовать старый SAS-токен. Вместо этого, перед каждой

новой попыткой загрузки файла из очереди, приложение должно выполнить запрос к своему бэкенду для генерации свежего, действительного SAS-токена для этой конкретной операции.[13] В очередь на отправку должен попадать не сам SAS-токен, а только метаданные, достаточные для его повторного запроса.

Эта ситуация представляет собой классический конфликт между требованиями безопасности и надежности. Лучшая практика безопасности (короткоживущие токены) напрямую противоречит требованию надежности (долгосрочные повторные попытки), если архитектура явно не предусматривает механизм их согласования. Вероятно, бэкенд и клиентское приложение не полностью согласованы в стратегии повторных попыток: бэкенд предоставляет токен, предполагая немедленную загрузку, в то время как клиенту необходимо обрабатывать загрузки, которые могут быть значительно отложены. Решение требует изменения логики клиента, чтобы сделать ее более «осведомленной» о состоянии аутентификации и необходимости обновления токенов.

Причина №3: Некорректная сериализация состояния задачи на загрузку

- **Вероятность:** Высокая (75%)
- **Основная гипотеза:** Приложение пытается сохранить всю задачу на загрузку для очереди повторных попыток путем сериализации объекта FormData с помощью JSON.stringify. Этот процесс неявно (без ошибок) отбрасывает двоичные данные файла, что приводит к тому, что повторная попытка выполняется с «пустым» телом запроса.

Детальный анализ

Несериализуемость FormData

Объект FormData — это не простой объект с данными в JavaScript. Это сложный

интерфейс, представляющий данные формы, включая двоичные объекты (Blob), который тесно связан с нативным сетевым стеком браузера или мобильной платформы.[14] Он не имеет осмысленного представления в формате JSON. Алгоритм

JSON.stringify обрабатывает только перечислимые (enumerable) собственные свойства объекта.[15] Внутренние данные

FormData, включая ссылки на файлы, не хранятся в таких свойствах, поэтому при сериализации они просто игнорируются.

Ограничение AsyncStorage и требование строки

Стандартный механизм для персистентного хранения данных в React Native, AsyncStorage (и его современный преемник @react-native-async-storage/async-storage), может хранить только пары ключ-значение, где значением является строка.[16] Это заставляет разработчиков сериализовать любой сложный объект в строку JSON перед сохранением, используя канонический подход:

JSON.stringify(value).[18]

Сценарий «тихого сбоя»

Здесь и кроется коварство этой ошибки. Когда разработчик вызывает JSON.stringify(myFormDataObject), это не приводит к возникновению исключения. Функция просто возвращает строку, представляющую пустой объект: '{}'. Эта строка успешно сохраняется в AsyncStorage. Позже, когда логика повторной попытки считывает и десериализует эту строку (JSON.parse('{}')), она получает валидный, но совершенно пустой объект. Последующая попытка извлечь данные файла из этого объекта (например, parsedObject.get('file')) вернет undefined, поскольку в десериализованном объекте нет никакой информации о файле.

Паттерн «Сериализуемый дескриптор задачи»

Правильный подход заключается в создании простого, полностью сериализуемого объекта JavaScript (дескриптора задачи), который содержит только необходимую *метаинформацию* для выполнения загрузки. Именно этот объект, а не FormData, должен сохраняться в персистентном хранилище.

Пример такого дескриптора:

JSON

```
{
  "jobId": "unique-id-123",
  "permanentFileUri": "file:///path/to/app/documents/image.jpg",
  "endpointUrl": "https://api.example.com/upload",
  "metadata": { "textEntry": "Текстовая запись от техника" },
  "retryCount": 0
}
```

Этот объект легко сериализуется в JSON, сохраняется и восстанавливается без потерь. При выполнении повторной попытки логика считывает этот дескриптор, находит файл по permanentFileUri, заново создает объект FormData и наполняет его данными из дескриптора перед отправкой. Совет из обсуждения на Reddit косвенно подтверждает этот подход: «Вы храните фактические данные изображения в redux? Если да, то в этом ваша проблема. Изображения должны храниться как файлы... и загружаться через нативный поток».[19] Это подразумевает хранение

ссылки (URI) в персистентном состоянии, а не самих данных.

Эта ошибка является тонкой, но чрезвычайно распространенной в системах, которые пытаются соединить сложные нативные API (такие как работа с файлами) с моделями сериализации JavaScript. Она идеально объясняет, почему код «работает в лаборатории» (где объект FormData может существовать в оперативной памяти в течение короткого теста) и «падает в поле» (где требуется

сохранение и восстановление состояния между сеансами работы приложения).

Причина №4: Потеря состояния в оперативной памяти из-за завершения процесса ОС

- **Вероятность:** Средне-высокая (65%)
- **Основная гипотеза:** Очередь загрузки и связанное с ней состояние управляются в волатильной оперативной памяти (например, в состоянии компонента React или в неперсистентном хранилище Redux). Когда приложение переходит в фоновый режим в полевых условиях, операционная система может принудительно завершить его процесс для освобождения ресурсов, что приводит к полной потере очереди.

Детальный анализ

Жизненный цикл мобильного приложения

В отличие от веб- или настольных приложений, мобильные приложения подчиняются строгому жизненному циклу, управляемому операционной системой. Приложение может быть в активном состоянии (active), фоновом (background) или полностью завершенном (killed).[20] ОС, особенно Android, может быть очень агрессивной в завершении фоновых процессов для высвобождения оперативной памяти, например, при поступлении входящего звонка или запуске другого ресурсоемкого приложения.[22]

Волатильность состояния React/Redux

По умолчанию все состояние, управляемое внутри компонентов React (useState,

useReducer) или в стандартном хранилище Redux, существует только в оперативной памяти приложения. Когда процесс приложения уничтожается, эта память освобождается, и все хранившееся в ней состояние безвозвратно теряется.[23]

Решение с помощью redux-persist

Промышленным стандартом для решения этой проблемы является библиотека redux-persist. Она автоматически сохраняет состояние хранилища Redux (или его части, определенные через whitelist или blacklist) в персистентное хранилище (например, AsyncStorage) при каждом его изменении.[24] При запуске приложения

redux-persist восстанавливает (rehydrates) состояние из сохраненной копии. Использование компонента PersistGate позволяет отложить рендеринг пользовательского интерфейса до тех пор, пока процесс восстановления не будет завершен, предотвращая отображение неконсистентных данных.[24]

Как это приводит к «undefined values»

Если очередь загрузки была потеряна из-за завершения процесса, пользовательский интерфейс при следующем запуске может все еще считать, что есть ожидающие загрузки. Это может произойти, если какой-то другой флаг (например, isUploading = true) был сохранен в AsyncStorage отдельно, а сама очередь — нет. При попытке отобразить статус загрузки, UI-компонент обращается к элементу очереди, например, queue. Но поскольку очередь теперь является пустым массивом (восстановлена в свое начальное состояние), queue возвращает undefined. Передача этого значения в дочерний компонент или функция для получения прогресса (getProgress(undefined)) вызывает зафиксированную в логах ошибку.

Эта проблема указывает на «неполную стратегию персистентности» в архитектуре приложения. Возможно, команда сохраняет некоторые пользовательские данные, но упустила из виду необходимость сохранять состояние выполняемых *операций*, таких как очередь загрузки. Это

распространенное упущение в приложениях, которые дорабатываются для поддержки офлайн-режима, а не проектируются с ним с самого начала. Требуется провести аудит всего состояния приложения, определить, что является эфемерным, а что должно переживать сбой приложения, и соответствующим образом настроить redux-persist.[24]

Причина №5: Ненадежное фоновое выполнение при использовании стандартного fetch

- **Вероятность:** Средняя (50%)
- **Основная гипотеза:** Механизм повторной загрузки реализован с использованием стандартных таймеров JavaScript (setTimeout/setInterval) и стандартного fetch API. Эти инструменты не предназначены для надежного, длительного выполнения, когда приложение находится в фоновом режиме, и часто подвергаются троттлингу (замедлению) или принудительному завершению со стороны ОС.

Детальный анализ

Ограничения JavaScript в фоновом режиме

Когда приложение React Native переходит в фоновый режим, операционная система резко сокращает ресурсы, выделяемые его JavaScript-потоку. Таймеры становятся ненадежными (могут срабатывать со значительной задержкой или не срабатывать вовсе), а длительные сетевые запросы, инициированные с помощью fetch, могут быть приостановлены или прерваны без предупреждения для экономии заряда батареи и памяти.[27]

Необходимость в нативных фоновых службах

Для настоящей фоновой загрузки файлов необходимо делегировать задачу базовой нативной операционной системе, у которой есть механизмы для управления длительными передачами данных независимо от жизненного цикла пользовательского интерфейса приложения. Библиотеки, такие как react-native-background-upload, созданы именно для этой цели.[27] Они работают, используя нативные API:

android-upload-service на Android и NSURLSession на iOS, которые являются официально поддерживаемыми ОС способами для выполнения отказоустойчивых фоновых передач.[27]

Обманчивость лабораторных тестов

В контролируемой лабораторной среде приложение почти всегда находится на переднем плане во время теста загрузки. В этих условиях fetch и setTimeout работают идеально. Расхождение проявляется только в полевых условиях, где переход в фоновый режим является обычным действием пользователя.

Как это приводит к «undefined values»

Сетевой запрос fetch, выполняемый в фоновом режиме, может быть прерван операционной системой на полпути. Промис, возвращаемый fetch, может никогда не разрешиться или не отклониться, оставляя приложение в «подвисшем» состоянии. В другом варианте, он может быть отклонен с общей сетевой ошибкой, к обработке которой логика приложения не готова. Блок catch, ожидающий структурированный объект ошибки (например, с полем error.message), может получить undefined, что и приведет к итоговой ошибке.

Это указывает на то, что приложение, вероятно, построено на предположении «foreground-first» (приоритет переднего плана). Архитектура не учитывает событийно-ориентированную, часто прерываемую природу мобильных приложений. Надежное решение должно делегировать длительные операции ввода-вывода нативному слою. Для приложения «полевого сервиса», которое по

своей природе работает в условиях отсутствующего или прерывистого соединения, архитектура должна быть изначально спроектирована как «offline-first» и устойчивой к фоновому режиму. Простое добавление цикла повторных попыток на JavaScript является недостаточной мерой.

Заключение и приоритизированные рекомендации

Синтез результатов

Проблема P1 является не единичной ошибкой, а системным сбоем, вызванным «каскадом волатильности». Текущая архитектура не устойчива к реальным условиям эксплуатации в поле. Сообщение об ошибке «undefined values» является общим симптомом для нескольких различных точек отказа в жизненном цикле данных: от первоначального захвата файла (Причина №1), аутентификации (Причина №2), сохранения состояния (Причины №3 и №4) и до непосредственного выполнения загрузки (Причина №5).

Приоритизированный план действий

Исправления должны внедряться в порядке убывания вероятности, так как устранение проблем на ранних этапах цепочки может нивелировать симптомы, проявляющиеся на последующих этапах.

1. **Немедленное исправление (Причины №1 и №3):** Внедрить паттерны «Копирование в постоянное хранилище» и «Сериализуемый дескриптор задачи». Это гарантирует, что данные для повторной попытки будут действительными и корректно сохранены. Это изменение окажет наибольшее влияние.
2. **Следующий шаг (Причина №2):** Модифицировать логику повторной попытки таким образом, чтобы перед каждой попыткой загрузки запрашивался новый SAS-токен.

3. **Архитектурное улучшение (Причина №4):** Внедрить redux-persist для редьюсера, управляющего очередью загрузки, чтобы гарантировать ее сохранность после завершения работы приложения.
4. **Долгосрочная отказоустойчивость (Причина №5):** Заменить логику загрузки на основе fetch на специализированную библиотеку для фоновых загрузок (например, react-native-background-upload), чтобы обеспечить завершение загрузок, даже когда приложение неактивно.

Сводная таблица для участников проекта

Причина	Вероятность	Краткое описание проблемы	Рекомендация высокого уровня
1. Недействительность эфемерных URI	90%	Попытка повторной загрузки использует временный URI файла, который становится недействительным после перезапуска приложения или очистки кэша.	Немедленно копировать выбранный файл в постоянный каталог (documentDirectory) и сохранять в очереди URI на эту постоянную копию.
2. Истечение срока действия SAS-токена	80%	Повторная попытка использует просроченный SAS-токен Azure, что приводит к ошибке аутентификации 403 Forbidden.	Перед каждой повторной попыткой загрузки запрашивать у бэкенда новый, действительный SAS-токен.
3. Некорректная сериализация FormData	75%	Попытка сохранить объект FormData в AsyncStorage через JSON.stringify приводит к потере данных файла.	Сохранять не FormData, а простой сериализуемый JS-объект (дескриптор задачи) с метаданными (URI файла, URL и т.д.).

4. Потеря состояния в оперативной памяти	65%	Очередь загрузки хранится в оперативной памяти (состояние React/Redux) и теряется при принудительном завершении приложения системой.	Использовать redux-persist для автоматического сохранения и восстановления состояния очереди загрузки в AsyncStorage.
5. Ненадежное фоновое выполнение	50%	Использование стандартного fetch и таймеров JS для повторных попыток ненадежно в фоновом режиме; ОС может прервать процесс.	Заменить fetch на специализированную библиотеку для фоновых загрузок (react-native-background-upload), использующую нативные API.

Works cited

1. react-native-image-picker uri doesn't persist for android - Stack ..., accessed July 23, 2025, https://stackoverflow.com/questions/55342899/react-native-image-picker-uri-doesnt-persist-for-android
2. react-native-image-picker - persistent storage after rebuild - Stack Overflow, accessed July 23, 2025, https://stackoverflow.com/questions/54221671/react-native-image-picker-persistent-storage-after-rebuild
3. [android][expo-image-picker] content:// URIs are not usable in ExponentImagePicker.launchImageLibraryAsync #24172 - GitHub, accessed July 23, 2025, https://github.com/expo/expo/issues/24172
4. FileSystem - Expo Documentation, accessed July 23, 2025, https://docs.expo.dev/versions/latest/sdk/filesystem/
5. How to Save Files to a Device Folder using Expo and React-Native | Farhan Says Hi, accessed July 23, 2025, https://www.farhansayshi.com/post/how-to-save-files-to-a-device-folder-using-expo-and-react-native/
6. FileSystem (next) - Expo Documentation, accessed July 23, 2025, https://docs.expo.dev/versions/latest/sdk/filesystem-next/
7. Unable to remove DocumentPicker files with FileSystem.deleteAsync · Issue #6335 - GitHub, accessed July 23, 2025, https://github.com/expo/expo/issues/6335
8. JavaScript: Upload image to Blob Storage - JavaScript on Azure ..., accessed July

23, 2025, https://learn.microsoft.com/en-us/azure/developer/javascript/tutorial/browser-file-upload-azure-storage-blob
9. Configure an expiration policy for shared access signatures (SAS) - Azure Storage, accessed July 23, 2025, https://learn.microsoft.com/en-us/azure/storage/common/sas-expiration-policy
10. Control access to private data using Azure storage shared access signatures - C# Corner, accessed July 23, 2025, https://www.c-sharpcorner.com/article/control-access-to-private-data-using-azure-storage-shared-access-signatures/
11. AZFD0006: SAS Token Expiring - Azure Functions, accessed July 23, 2025, https://docs.azure.cn/en-us/azure-functions/errors-diagnostics/diagnostic-events/azfd0006
12. Getting a dataset corrupt error while training a model in azure vision studio :Failed to authenticate: SAS token is invalid or expired. - Learn Microsoft, accessed July 23, 2025, https://learn.microsoft.com/en-us/answers/questions/1377505/getting-a-dataset-corrupt-error-while-training-a-m
13. How can I handle Azure SAS Expiration? - Stack Overflow, accessed July 23, 2025, https://stackoverflow.com/questions/14873336/how-can-i-handle-azure-sas-expiration
14. Using FormData Objects - Web APIs | MDN, accessed July 23, 2025, https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest_API/Using_FormData_Objects
15. JSON.stringify() - JavaScript - MDN Web Docs, accessed July 23, 2025, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify
16. AsyncStorage - React Native, accessed July 23, 2025, https://reactnative.dev/docs/asyncstorage
17. A guide to React Native's AsyncStorage - LogRocket Blog, accessed July 23, 2025, https://blog.logrocket.com/guide-react-natives-asyncstorage/
18. Usage | Async Storage - GitHub Pages, accessed July 23, 2025, https://react-native-async-storage.github.io/async-storage/docs/usage/
19. Suggestions on implementing upload queue, resumable after being offline and can survive an app crash. : r/reactnative - Reddit, accessed July 23, 2025, https://www.reddit.com/r/reactnative/comments/sre6r0/suggestions_on_implementing_upload_queue/
20. AppState - React Native, accessed July 23, 2025, https://reactnative.dev/docs/appstate
21. Understanding App States in React Native: Active, Background, and Killed - Medium, accessed July 23, 2025, https://medium.com/@vectoreman67/understanding-app-states-in-react-native-active-background-and-killed-c746afbdb554
22. How do I persist state for android apps killed in the background in react-native, accessed July 23, 2025,

https://stackoverflow.com/questions/42350708/how-do-i-persist-state-for-android-apps-killed-in-the-background-in-react-native
23. Preserving and Resetting State - React, accessed July 23, 2025, https://react.dev/learn/preserving-and-resetting-state
24. React Native Coach, accessed July 23, 2025, https://blog.reactnativecoach.com/the-definitive-guide-to-redux-persist-84738167975
25. How To Use Redux Persist in React Native with Asyncstorage - Jscrambler, accessed July 23, 2025, https://jscrambler.com/blog/how-to-use-redux-persist-in-react-native-with-asyncstorage
26. Using Redux-Persist and AsyncStorage in React Native | by JoAnna Park | Medium, accessed July 23, 2025, https://joannabpark1.medium.com/using-redux-persist-and-asyncstorage-in-react-native-9ea115b4635c
27. Vydia/react-native-background-upload: Upload files in your ... - GitHub, accessed July 23, 2025, https://github.com/Vydia/react-native-background-upload
28. react-native-background-upload - NPM, accessed July 23, 2025, https://www.npmjs.com/package/react-native-background-upload