Tech Feasibility - 10/22/2025

Shark Bytes

Clients: David Rogowski, Mara Dzul, Pilar Rinker

Mentor: Scott LaRocca

Team: Jered Angous, Daniel Arden, Alexander King, Anthony Narvaez

The purpose of this document is to discuss how our team has weighed and selected the different technologies that will comprise the tech stack of our application.

Table of Contents:

0.0: Introduction

Over 40 million people across seven states depend on the Colorado River and its surrounding ecosystems to sustain them, making it one of the most vital resources in North America. Yet, this critical resource faces increasing pressure from climate change, droughts, and overuse. Understanding the health and status of this area not only helps conserve the keystone species that call these areas home, but also serves as a marker for the overall health and wellbeing of the millions that count on these waters. Organizations like the U.S. Geological Survey, the Arizona Game and Fish Department, the Grand Canyon Monitoring and Research Center, and the U.S. Fish and Wildlife Service are at the frontlines of this effort, collecting data that helps inform conservation efforts, identify environmental stressors, and guide resource allocation.

However, the current data collection process used by our sponsors, members of these same organizations, relies heavily on an antiquated program known as SHOALS. This critical application initially allowed researchers to efficiently transfer information from the fish they caught to a central database for analysis. While still critical for the research effort, SHOALS has been unmaintained for well over six years. This, combined with its unintuitive user interface, total lack of documentation, and its dependency on the outdated programming language Visual Basic, has made SHOALS increasingly difficult to use. Biologists now spend valuable time deciphering control files and determining which of the many executable files they need to use. It is clear that there is a need for a more modern and reliable approach.

Our team is building a newer, more flexible knowledge entry system to solve these operational challenges. We aim to provide an intuitive, easy to navigate program that improves on the core benefits of SHOALS while being well documented, easily maintained, and reliable for use in the field. Our app will provide:

- A modernized user interface and experience
- More flexibility in defining data entry fields
- Improved statistics and graphical data representation
- One consolidated application that fits the needs of multiple organizations

Ultimately, this application will streamline the data collection process for biologists and researchers across many agencies, reduce time wasted figuring out how to use the program, and ensure that our sponsors continue to provide the critical data that these ecosystems, and entire communities, depend on.

1.0: Technological Challenges

1.1: Introduction:

Making a flexible data entry app with mission-critical data integrity, reliability, and performance is a difficult task. Our team is currently considering the numerous technical challenges we must solve to make this a reality:

1.2: Technological Challenges

2.1: App Framework

- Create and render an interactive user interface that works across multiple platforms.

2.2: Field Definition Language

- Define the names, data types, and constraints of the user's required fields.

2.3: Local Data Storage

- Store the user's inputted data locally.

2.4: Last Catch History

- Store a subset of information of past entries for comparison.

2.5: Automatic External Backups

- Automatically back up the entries to an external hard drive.

2.6: Bluetooth PIT Tag Reader

- Communicate with external Bluetooth PIT tag readers.

2.0: Technological Analysis

In this section, we analyze a variety of options for addressing our technological challenges. We outline our desired characteristics, provide an overview of available options, and then weigh the pros and cons of each one, and select the option we believe will best fit our project's use case.

2.1: App Framework

2.1.0: Introduction

Our project requires an app framework to provide an interactive user interface, multi-platform support, reduced code complexity, and better code maintainability. An app framework enables better code organization, better reliability, and less complexity than building a solution from scratch. By utilizing an app framework that provides us a basis for our code, we can focus more on providing the clients with their required functionality than building an entire UI rendering solution from scratch.

2.1.1: Desired characteristics

We desire an app framework that supports desktop cross-platform (primarily Windows and Linuxs), that has solid performance and reliability. It should have efficient memory usage and a small bundle size so it can run on outdated computers with limited memory.

The ideal solution would be cross-platform, have well-maintained libraries for interacting with tools like databases, compression algorithms, and bluetooth support. It should be performant, reliable, display a consistent UI across platforms, and be simple to use. Having a strong community and strong backing is a plus for guaranteeing future support for the underlying language and libraries. The front-end and back-end languages should be easy to write and not have a steep learning curve (this rules Tauri out due to Rust's notorious complexity). Additionally, it should have a compact bundle size and efficient memory usage so the app can run well on older computers.

2.1.2: Alternatives

2.1.2.1: Electron

Electron is the most mature option for creating cross-platform desktop applications, used by major applications such as Visual Studio Code and Discord. It uses HTML, CSS, and JavaScript for the frontend, and React for the backend. However, Electron apps struggle with a large bundle size, unoptimized performance due to the reliance on React, and high memory usage due to relying on a Chromium browser for rendering. Electron apps are also sometimes more prone to crashing due to the large memory requirements, reducing reliability. Even with these disadvantages, Electron is very mature, has major backing, a large community, and is likely the simplest of the three frameworks to use as a developer.

2.1.2.2: Tauri

Tauri is the newest solution of the bunch. It is similar to Electron apps in that it relies on HTML, CSS and JavaScript for coding the frontend. Where it is different from Electron is that, instead of bundling a Chromium browser with each app, it uses the OS' native WebView, significantly reducing bundle size and increasing performance. Due to its use of Rust on the backend, it is very performant, reliable due to memory safety, and has excellent memory usage if memory is handled correctly. However, Rust also significantly increases the learning curve of using Tauri. Additionally, Tauri is new and lacks major backing, introducing the risk of major breaking changes or potentially becoming unmaintained if the open source community can't keep it going.

2.1.2.3: Flutter

Flutter is cross-platform, has great performance and consistency due to bundling its own rendering engine, and has major backing from Google. Syntactically, the Dart language is like a hybrid of JavaScript and C, which is not the simplest but has a lower learning curve than Rust. Additionally, Flutter is used by many apps, is mature, and has a large community. Flutter also has existing libraries for interacting with CSVs, SQLite databases, JSON, and Bluetooth, all of which may have a place in our project.

2.1.3: Analysis

2.1.3.1: Electron

Electron is currently the most prominent cross-platform application framework. However, Electron bundles an entire Chromium engine in with every application. This results in an excessively large bundle size, and inefficient RAM usage. Inefficient RAM usage can slow down computers and even cause the program to crash on computers with limited RAM. We are concerned that these downsides would make running an Electron application on older government computers sluggish and unreliable. With these downsides in mind, our team decided against using Electron for our project.

2.1.3.2: Tauri

After a brief discussion, our team decided we did not want to use Tauri. The lack of major backing for the framework, and small size of the community surrounding it makes it difficult to trust the framework will continue to be supported. Additionally, with Tauri being so new, we doubt the maturity of Tauri and are concerned about new updates bringing breaking changes. Lastly, we evaluated Rust, Tauri's backend language. The steep learning curve of Rust could slow progress on our application - a major challenge when working in such a short timeframe.

2.1.3.3: Flutter

Flutter is maintained by Google and is used to power many high-profile applications. While most Flutter applications are designed for mobile devices, desktop applications received full stable support in 2022. Flutter is powered by Dart, a compiled, type-safe and memory-safe language, which ensures reliability and performance. Dart, while more complex than writing JavaScript as used in Electron, is significantly simpler to write. This combination of characteristics - simplicity, reliability, and major backing, make Flutter attractive to our team.

2.1.4: App Framework Chart			
	Flutter	Electron	Tauri
Desktop cross-platform	✅	✅	✅
Mobile cross-platform	✅	❌	🟡
Front-end language	Dart	HTML, CSS, JS	HTML, CSS, JS
Back-end language	Dart	JS	Rust
Performance	Great	Poor	Excellent
Reliability	Excellent	Varies	Excellent
Consistency	Excellent	Excellent	Varies
Simplicity	Moderate	High	Moderate
Bundle Size	Moderate	Very Large	Very Small
Memory Usage	Great	Poor	Excellent
Maturity	High	Very High	Moderate
Community	Large	Very Large	Moderate
Backing	Very strong	Strong	Weak
Open-source	✅	✅	✅

2.1.5: Chosen approach

Our team has decided to go with Flutter. It has a happy medium of performance, reliability, and ease of use. Dart's syntax being reminiscent of familiar languages such as C and JavaScript, and not requiring manual memory handling, significantly reduces the learning curve, making it possible for us to get our project running. The large community, major backing, maturity, and amount of large projects that use Flutter, mean it is likely going to be supported for a long time, making our project future proof. Additionally, the smaller memory requirements and bundle size make it likely our app will perform well even on outdated computers.

2.1.6: Proving Feasibility

We wrote a small command line script in Dart (Flutter's underlying language) that connects to a SQLite database, queries it for its schema, and prints out the names of the tables and the details of each attribute. This showcases Dart has the ability to connect to a database and reformat data. The next steps for proving feasibility will be to generate a visual form with fields from the database tables, retrieve the user's entered data, and enter them into the database.

2.2: Field Definition Language

2.2.0: Introduction

A field definition language describes the syntax used to describe the names, data types, constraints, and order of every field in the form. A robust field definition language will make our application customizable enough to fit a variety of use cases, rather than tailoring it for specifically aquatic research.

2.2.1: Desired Characteristics

We desire a field definition language that has the **maintainability** and **consistency** of a SQL database, with the **flexibility** and **extensibility** of JSON when needed. We also desire functionality for defining basic constraints, required fields, and predefined autocomplete suggestions. Additionally, we want a system that will scale well, a solid developer ecosystem, and reasonable simplicity so the structure is not difficult to interact with.

The biggest technical challenge we face is how to structure the fields our application should render. The clients requested support for customizing the names, order, and data types of fields, imposing basic constraints on different fields, and automatic population of common responses. Figuring out how to maintain high data integrity and reliability while keeping our application flexible enough to adapt to different use cases is at the top of our list.

The ideal solution would be easy to read, easy to write, and easy for the program to parse. It must strike the correct balance between flexibility and structure. Additionally, it must support basic constraints such as required fields. Furthermore, it should be reorderable, and support auto-suggestions for fields as specified by the user.

2.2.2: Alternatives

2.2.2.1: SQL

SQL is a language designed for querying databases. SQL originated in the 1970s and is still in use today. While SQL is mostly used for interacting with existing databases, it also has its own syntax for creating and editing databases via CREATE and ALTER statements. Additionally, many SQL databases have commands such as SHOW and PRAGMA to retrieve information about the structure of the database. This information could be used to define the names and datatypes of fields to present in our application.

2.2.2.2: JSON

JSON is a more general purpose format for defining data. JSON has been around since 2001 and is commonly used with JavaScript as a way to flatten JavaScript objects to a text format which can be stored on a hard drive. JSON is used in some static site generators such as Hugo and Eleventy to define the information presented on the rendered HTML page, and is also used in some applications such as VSCode to define the settings. Due to its flexibility, JSON can be used for almost anything, including a language for defining the fields of our app. However, JSON's flexibility also means a lack of structure, which could make parsing the exported data difficult.

2.2.2.3: CSV

CSVs are a common format for exporting structured data such as spreadsheets into a more portable, compact, and easy to parse format. The concept of CSVs have been around since the 1970's, being used in the IBM Fortran compiler. CSVs are very easy to work with, with libraries existing in most programming languages for importing and exporting data. However, CSVs are very rigid, designed typically to store only one table in a consistent array of rows and columns. This makes modeling complex relationships difficult.

2.2.3: Analysis

2.2.3.1: SQL

To test SQL for our application, we wrote a script which queries the database using a PRAGMA table_list statement to retrieve a list of the tables in the database. After this, the program runs PRAGMA table_info statements to retrieve a list of the current table's attributes. This provides us with information about every attribute, including the name, data type, and whether that attribute can be left NULL. By utilizing a SQLite database and pragma statements, a SQLite database schema can be used as a simple field definition language. However, it does lack functionality for changing the order in which fields appear, and autocomplete suggestions.

2.2.3.2: SQL + JSON

Pairing SQL with JSON could enable JSON to fill in the gaps SQL can't fill, namely reordering fields and autocomplete suggestions. We intend to experiment with creating a SQL table to handle autocomplete. This table could be named __autocomplete would work by using the names of attributes which require autocomplete as the primary key, and a corresponding JSON string stored in each row with the autocomplete suggestions necessary for that field. When the user clicks into an autocomplete field, the app will query the __autocomplete table, retrieve this JSON string of suggestions, parse it, and use it for the autocomplete suggestions. For handling the reordering of fields, we plan to create a __metadata table which stores a JSON string containing the attribute names in the order they should appear in the application. When the app is launched, it will query this table, parse the JSON, and render the fields in that order.

2.2.3.3: JSON

After discussing the structure of the main database our clients use, we've come to the conclusion JSON may be too flexible, and thus decided not to continue with the analysis of this option. Our clients' existing program, SHOALS, exports to a CSV which is then formatted and imported into a large relational database. JSON would enable too much flexibility in adding or removing attributes, which may compromise the integrity of our data. This could make it difficult to parse and reformat for importing into their main relational database.

2.2.3.4: CSV

After doing some preliminary research, we realized using CSVs for the field definition language would be difficult. We would need to create our own constraints syntax and handle all of this in the app logic, which may not prove reliable. It would also be difficult to model complex relationships such as foreign key relationships in relational databases. Therefore, CSVs as a field definition language do not make practical sense for our field definition language.

2.2.4: Field Definition Language Chart

	SQL	SQL Hybrid	JSON	CSV
Flexibility	Low	Moderate	High	Very low
Consistency	Excellent	Excellent	Moderate	Poor
Maintainability	Very good	Very good	Moderate	Poor
Extensibility	Low	High	High	Very low
Simplicity	Moderate	Moderate	High	Very high

2.2.4: Field Definition Language Chart

Required fields	Yes	Yes	App logic	App logic
Constraints	Excellent	Excellent	App logic	App logic
Foreign keys	Yes	Yes	No	No
Ecosystem	Very large	Medium	Large	Large
Developer experience	Familiar and Structured	Blended	Accessible	Simple
Security	Excellent	Very good	Moderate	Poor
Versioning	Standardized	Blended	External	Manual
Scalability	Excellent	Excellent	Varies	Poor

2.2.5: Chosen Approach

Our chosen approach is a SQL Hybrid with JSON, utilizing SQL for its defined structure and constraints engine, and JSON for flexibility and additional functionality such as field autosuggestions and storing metadata for how to render the SQL attributes as fields. This gives our program a defined structure that has high consistency and maintainability, with the flexibility of JSON when needed.

2.2.6: Proving Feasibility

We wrote a Dart script which queries a SQL database to get a list of its tables, then queries each table for its attributes. It then stores all of these details in different Dart objects - a database object with a list of Table objects, and those table objects contain a list of the Attribute objects. The next steps for proving feasibility will be to get user input and enter the inputted data from the fields into the database. Steps after that include setting up autocomplete suggestions by storing JSON in a generated autocomplete lookup table, and finding a way to store metadata, potentially in the database or a separate JSON file.

2.3: Local Data Storage

2.3.0: Introduction

Managing data locally is a critical component of the application's functionality. The system must store and retrieve information effectively while maintaining portability, reliability, and ease of access. Choosing the right local data storage solution will determine how effectively data can be saved, updated, and backed up externally. Selecting an appropriate format would directly impact development efficiency, performance, and long-term maintainability. This section examines potential solutions for local data storage and evaluates which options best meet the project's requirements.

2.3.1: Desired characteristics

The ideal local data storage format should be portable, it should be easily converted to other formats if needed, and it should be moved and backed up to external hard drives. Ease of use is another characteristic; the storage should be straightforward to set up, access, and modify during development and runtime. Flexibility is necessary to adapt to various data structures depending on needs, and reliability to ensure data is preserved during reads, writes, and interruptions. It should be compact so the computer's storage doesn't fill up quickly.

2.3.2: Alternatives

2.3.2.1 SQLite

SQLite is one of the leading solutions. It is an open source SQL database engine that is widely used for systems that require lightweight relational storage. SQLite stores the entire database in one file, so it can be highly portable and easy to back up. It supports queries, indexing without requiring a separate server. Its design is ideal for applications that rely on structured data and need fast, reliable, and compact storage.

2.3.2.2 CSV

CSV is one of the simplest and most common local data formats. It is often used to import or export tabular data between databases. Each row represents a record, and columns are separated by commas. CSV files are plain text, so they are very portable. They can be easily edited with text editors or spreadsheet software. The simplicity comes at the cost of flexibility; CSV cannot store hierarchical data, and large files can be very repetitive and bulky due to being plain text.

2.3.2.3 JSON

JSON is a lightweight, text-based format that stores data as key-value pairs and supports nested and hierarchical structures. JSON is very portable, since it is stored as plain text and supported by many modern languages. Its flexibility allows it to represent complex data structures, making it ideal for an application like ours. JSON is easy to parse and modify. Although it becomes harder to read as datasets become larger and more deeply nested.

2.3.3: Analysis (needs to be rewritten - talk about how things were ruled out, talk about testing done.)

2.3.3.1: SQLite

To test SQLite, we created Dart scripts that inserted, deleted, and modified data within a local SQLite database. These tests helped evaluate how easily SQLite could integrate with our application and how efficiently it handled repeated operations. SQLite demonstrated the strongest overall performance. It was compact and reliable, using a binary storage format that minimized file size and improved speed. Its single-file design made it portable and easy to back up. Although it required SQL commands for setup, this structure offered consistency and control for data management. The relational model made it easy to organize and query data efficiently. No errors or slowdowns were observed during testing, and data integrity was preserved. Because it met all of our desired characteristics portability, flexibility, compactness, and reliability SQLite was selected as the best option moving forward.

– It does have cons, as the file is not human-readable. A little higher learning curve, as it uses SQL commands, but that could be useful, so users don't try to change the structure. Manual data correction would have to be done using SQL commands; you cannot just change the file.

2.3.3.2: CSV

CSV was initially appealing for its simplicity and human readability. Our tests proved it is straightforward to create and modify data files manually. As we added and removed entries, it became harder for CSV to be used in an effective and efficient manner. It did not perform as well as the other methods; it requires larger file sizes, which is not ideal, and well as limited flexibility. CSV is good for smaller tabular data, but not a fit for our needs. It stands out for its simplicity and human readability, but cannot manage complex data or relationships. Based on these results, CSV was ruled out because it could not handle the scalability or data complexity required by the project.

2.3.3.3: JSON

JSON provided strong flexibility and compatibility with Dart; it was easy to modify. The data files became harder to maintain as the data sets would grow larger or more complex. Without built-in indexing or query support, performing frequent searches or updates required more custom code and processing time. While JSON's hierarchical structure offers benefits for representing complex data, it lacks the efficiency and performance needed for continuous read-write operations. These limitations made JSON less practical for our application's needs, leading us to prioritize SQLite's structured querying and reliability instead. After testing and doing research, we ruled out JSON because, while flexible, it lacked the reliability and performance needed for continuous database operations.

2.3.4: Local Data Storage Chart			
	SQLite	CSV	JSON

2.3.4: Local Data Storage Chart			
Ease of use	Excellent	Excellent	Excellent
Portability	High	Medium	Medium
Flexibility	High	Low	High
Size	Small	Medium	Medium
Human readability	Low	High	Medium

2.3.5: Chosen Approach

After consideration, SQLite is the best approach for us. Other sections of our application will also use it, making it easier to interact with data without having to convert data files to other formats. It is efficient, reliable, flexible, and stores data in a compact format. Additionally, it is portable, and can be easily backed up or moved across systems. SQLite is also supported by many programming languages. Overall, SQLite provides the best balance between the desired characteristics we want for the local data storage in our application.

2.3.6: Proving Feasibility

We have made tests for SQLite and CSV using Dart to import databases, call tables, and table cells. We will make more tests to save entries to a database, remove entries, and make changes in dart to ensure SQLite functions all work for our desired needs in the program. Tests for moving data from different databases to be saved or converted to other formats to ensure backups can be made. Once our app has input fields, we will have tests to ensure those fields work correctly, read, and are saved to storage.

2.4: Storing last catch history

2.4.0: Introduction

One of the features requested by our clients was the ability to instantly view the information from a fish's previous capture when its PIT tag is scanned. This feature ensures that researchers can quickly compare new data entries with the last recorded data. If a significant difference is detected by the system, it will prompt the user to verify the entry before saving the information. To fix this issue, the application needs a reliable and compact way of storing the last catch history for every fish that has a PIT tag. This dataset will be accessed frequently because researchers catch and scan fish constantly throughout their field expedition. So speed, reliability, flexibility, and data integrity are important. The system should never lose or corrupt previous entries, even if the system has an unexpected power loss or the application crashes. Overall, the solution to this issue must balance speed, compactness, reliability, and maintainability.

2.4.1: Desired characteristics

To effectively store and retrieve the latest catch entries for each fish, the solution must balance between performance, efficiency, and reliability. The system must have high speed access to quickly retrieve the previously recorded entries when a pit tag is scanned ensuring smooth workflow. Since the field laptops used have limited storage capacity the ability to represent thousands of entries using minimal disk space is important. It must also provide quick reading , updating, and writing of entries with low overhead so the application has a smooth performance on less than ideal hardware. Data integrity is a top priority; the information collected must not be lost or corrupted even if the system breaks down. Finally, the solution should support extensibility to handle future data fields that will be created. An ideal solution would be a lightweight, embedded database that is fast, reliable, low storage, strong data integrity, and is scalable for the future.

2.4.2: Alternatives

2.4.2.1: JSON

JSON is a human readable data format that is usually used for storing structured data. An advantage to using JSON is that it's simple to use and can be read and written without much setup. One downside of JSON is the fact that it is not optimal with large datasets, as the number of data grows, the whole file becomes slower, which is not optimal for the application we are creating. Another disadvantage with JSON is the fact that it does not have built-in protection against corruption, which could occur with researchers in the field.

2.4.2.2: CSV

CSV is a widely used text-based format for simple data. The main advantages are the minimal storage overhead and great universal compatibility, which makes them easy to transfer and back up. The disadvantage of CSV is that it has no protection against concurrent access or data corruption, which causes problems.

2.4.2.3: SQLite

SQLite is an embedded relational database that stores data in a single file while also providing full SQL query capabilities. It is reliable, fast, and compact, which makes it perfect for offline applications like data collection. SQLite databases are also highly portable, need no setup, automatically recover in case of a crash, and can handle thousands of entries. The main disadvantage is the larger footprint and higher overhead, which is not as bad as the other alternatives.

2.4.2.3: Custom Binary Format

A custom binary format stores the data in a structured format that is not readable to humans. This approach is very fast and compact because it's written as the program would expect it, both minimizing overhead and parsing time. The challenge with this alternative is that for the future, if there are any maintenance issues, the future developer would need to understand the binary layout to make changes. A fix to this is making the application open source, which we plan to do so this alternative is a viable option.

2.4.3: Analysis

Each storage method had its desired characteristics evaluated based on the speed of access, compact data size, ease of access, low overhead, data integrity, and extensibility. The PIT tag scanner has not yet been delivered, so full-scale testing is not possible. For now, small scale prototypes are being used to check performance and data handling until real field data becomes available..

2.4.3.1: SQLite

Initial testing shows that SQLite provides the most balanced performance. It has fast indexed queries which allow quick lookups and its single file design ensures a compact and reliable storage. It also supports future scalability, making it the leading candidate at this stage.

2.4.3.2: CSV

CSV was tested since it is currently used in the sholes system and for its simplicity and portability. While CSV offers low overhead and small file sizes, its lack of protection against data corruption make it unsuitable for field studies. Therefore, our team has decided CSVs are unsuitable for our application.

2.4.3.3: JSON

JSON proved easy to read and add but as larger datasets are introduced the performance decreases drastically. Another disadvantage is that JSON has no built in data protection, which increases the risk of corruption. Hence, our team has ruled JSON out.

2.4.3.4: BIN Format

The custom binary format showed the fastest access speed and the smallest data footprint. However, it has been ruled out due to its high maintenance difficulty and lack of human readability. Making the format open source would help mitigate this issue by allowing community support, which we plan to do, but it does not completely remove the inherent complexity of maintaining a custom system. Due to this complexity, our team has decided to avoid creating a custom BIN format.

2.4.4: Storing Last Catch History Chart				
	SQLite	CSV	JSON	Bin file + custom format
Speed of access	Fast	Slow	Moderate	Very Fast
Compact data size	Large	Small	Very Large	Very Small
Ease of access	Good	Good	Good	Bad
Overhead	Medium	Low	Medium	Low
Data integrity	Very Good	Bad	Decent	Very Good
Extensibility	Very Good	Bad	Very Good	Decent

2.4.5: Chosen Approach

After evaluating all of the options, our team has selected SQLite for storing and retrieving the last catch history. It provides fast lookups, compact data storage, and excellent data integrity. SQLite also has a single file structure which minimizes storage needs while maintaining stability, even if an unexpected shutdown occurs. Another key advantage to this approach is that it offers cross platform compatibility and simple integration into existing field applications. This makes SQLite the most practical choice for long term data collection.

2.4.6: Proving Feasibility

At this stage we have not been able to fully test the SQLite integration since the pit tag and the scanner have not been provided. However, SQLite is proven to be a highly effective solution for data storage, offering quick and reliable access to the data. Once the hardware is available, prototype testing will be conducted to test the performance, stability, and data handling at scale. Based on SQLite's strong track record and initial tests, we are confident it will meet the project's requirements.

2.5: Automatic External Data Backup

2.5.0: Introduction

A crucial aspect of the data collection that takes place in the Colorado River is robustness and redundancy. Computing and data storage devices used here are at high risk of getting damaged or completely lost both in and around the river. To mitigate the probability of losing data while out on the field, the program should have the ability to automatically back data up to an external hard drive as it is being collected.

2.5.1: Desired Characteristics

The ideal automated backup system must meet several requirements. First, it must store data in an easily exportable format. This allows biologists to easily interpret and parse data for importing into the main database. The system must also back the data up as the data is being collected. This allows any incidents that may damage the laptop to not affect the data that has already been collected. The data needs to be backed up both the internal drive of the laptop as well as external flash drives. This prevents data loss when one of the data storage methods fails. The system should allow for automatic integrity checks on the data to detect and prevent errors in the data.

2.5.2: Alternatives

2.5.2.1: File Mirroring

File mirroring ensures that any time a file is written to or updated, it is immediately duplicated to the backup locations. In this case it would be duplicated from the main data file to the flash drive. This is a relatively simple method to implement as it can take advantage of built in Operating System tools like rsync to provide a real-time backup for any file type. However it doesn't do any error checking on its own.

2.5.2.2: Transactional Logging

Transactional Logging separates the main data from the data being entered by means of a log file. Data being entered is written into a log file which can then be verified and copied from there to various backup locations. In our case, it would be copied from the log file to the main data file in the local drive as well as the external flash drive. The main benefit of a system like this is that it is highly robust to corruption and events like a loss of power as the working data is completely separate from the stored data. Although the implementation might require slightly increased complexity, its resilience is a sought-after feature.

2.5.2.3: Database-Based Backups

Database-based Backups rely on the structure of database systems like SQLite to store data in an organized and easily accessible manner. In our case we would use database triggers to automatically backup the database from the main memory to the external flash drive. This approach is highly compatible with the method of data storage we are likely to use in the rest of the application, that is, SQLite. This system's automatic and consistent backup services provide a reliable backup method at the expense of slightly increased overhead.

2.5.2.4: Dual-Write

Dual-Write is a simple approach in which the data is written to both files on the internal and external drives simultaneously. It is relatively simple to program and perfectly redundant in real time, but requires very significant and attentive error-handling. While simple, this method leaves a lot to be desired. This includes error handling, data consistency with the main app, and inefficiencies when writing certain file types to the flash drive.

2.5.2.4: Snapshots

The Snapshots approach creates complete copies of entire directories. These are then written separately to both the internal and external drives. These backups can be taken periodically, whether by measure of time or of entered data, to ensure that the backup is always up to date. While a very thorough method, copying entire directories might add significant overhead that isn’t merited by the data being collected. It's also not exactly as real-time as some of the other methods.

2.5.3: Analysis

With these approaches in mind, we will analyze which solution, or combination of solutions, best fit the requirements for this module.

2.5.3.1 File Mirroring

We need our data backup system to be reliable in the field, efficient, and robust to corruption. File mirroring is a solid approach to this problem due to its simplicity, complete backups on both local and external memory drives, and dependency on operating system tools. However, this approach comes with drawbacks. File mirroring involves copying the entire contents of a directory to the external drive, which means overwriting entire files. This, in turn, results in inefficiencies in the program that would be drastically amplified as the size of the data increased. Further, the lack of error checking methods associated with this approach make it an undesirable method.

2.5.3.2 Transactional Logging

Transactional logging is an interesting approach as it is highly robust to corruption. Instead of directly writing to the main data file, this method creates a sort of “staging” file where new records are placed and of which the main and backup data files are updated from. That way, if there is a loss of power, or catastrophic event (like dropping the laptop in the river), the main data file and backups aren’t corrupted, only the logging file would need to be checked. The only issue with this method is that there might be compatibility issues with the database system that is intended to be used with the rest of the app. However, even despite this potential shortcoming, the emphasis on corruption protection and its compatibility with error checking make this an approach worth considering.

2.5.3.3 Database-based Backups

This leads straight into database-based backups. This method is specifically designed to use pre-existing database systems like SQL or SQLite, which means the built-in backup and replication services that come with these systems can be leveraged with little added complexity. The main database file we would use to store the last catch information could easily be updated while also providing backup functionality under the same technology. While there is some minimally increased overhead with this approach, the increased consistency with how the data is stored combined with the ease of use of the backups system in SQLite might make this a worthwhile investment.

2.5.3.4 Dual-Write

The Dual Write method is relatively straightforward as it only requires the module to simply write to both the main data file and to the backups at the same time. However, despite the ease of use, there are a lot of other considerations that need to be made to have a dual write system that is also reliable and provides atomicity, that is, protection against corruption after crashes, unexpected removals, or loss of power. This results in a system that, by itself, is too simple for our use case. The importance of redundancy and robustness cannot be overstated, especially when data collection trips can cost upwards of $80,000.00. A backup system that ensures proper data protection is crucial.

2.5.3.5 Snapshots

On the other hand, methods like the snapshots system are a little too complex for the application we are working on. While snapshots would indeed provide very thorough backups, we do not need full snapshots and history of the data that is being collected. A system like this would add significant overhead that would only result in unnecessary complexity. What we need is a lightweight and simple backup system that also provides protection against data corruption. Clearly, a hybrid approach is the way to go.

2.5.4: Automatic External Data Backup Chart						
	File Mirroring	Transactional Logging	Database Based Backups	Dual-Write	Snapshots	Hybrid
Easily Exportable Format	🟡	🟡	✅	🟡	❌	✅
Real Time Backups	✅	✅	✅	✅	❌	✅
Must Support Automatic Integrity checks	❌	🟡	🟡	🟡	🟡	✅
Local and External Backup	✅	✅	✅	✅	✅	✅

2.5.5: Chosen Approach

The approach that will best fit our application is one that combines the positive aspects of Transactional Logging, Database-Based Backups, and Dual Write. That is, a system that separates the verified and stored Main Data and backups from the data that is being scanned and entered by the biologists, saves a copy of the data to both the local drive and the external flash drives, as well as maintains the efficiency and consistency of using the SQLite database system throughout the app.

In order to achieve this, the chosen approach works by maintaining a log of the data that is being entered by the biologists in a separate file. The information in this file is in the .sqlite format used by SQLite for both consistency, and efficiency when writing to other files. This log file is used as a template from which data is verified and written to both the main database file and the external backup. Backup writes will be done every couple scans which, when combined with the efficiency of the .sqlite file format, allows for efficient and real time backups.

What is particularly interesting about this method is how much of it SQLite handles by itself already. For example, transactional logging is already a core aspect of writing to SQLite databases. This, combined with the use of database transactions, results in a fast and efficient write that can be duplicated onto many database files. This method also scales very well as the number of data records increases thanks to the short time complexity of inserting into a database rather than duplicating entire data files.

2.5.6: Proving Feasibility

To test the chosen approach we decided to simulate the scanning of fish by means of generating random fish entries. This was done in a short Dart program that allowed the user to enter the name of the drive they wanted to store backups in as well as the number of fish entries they wanted to store. The program would then create a database file and directory if it didn't already exist, create the database tables, and insert the entries into both the local and external database files. The testing revealed that backing data up to external drives was not only feasible but efficient and practical. It allows data to be easily exportable, and even works on multiple flash drives for even greater redundancy.

2.6: Bluetooth Communication With PIT Tag Reader

2.6.0: Introduction

The organizations using this app use an RFID scanner that reads the ID number of PIT tags on fish. These scanners connect to the app via bluetooth, and transmit the data to the program. This is essential for efficient retrieval and updating of specific data. This means having a method by which the app communicates with these scanners is critical for providing the value proposition.

2.6.1: Desired characteristics

The ideal communication method between the laptop and the scanner must meet several requirements. Namely, it must be a reliable method that allows for automatic signal recovery. It should also support a large variety of scanners and firmware versions, allowing researchers to focus on collecting data and not on the scanner. It is also important for the communication system to support Bluetooth Classic and Serial Port Profiles (SPP). This is because many of the scanners used by our client use this flavor of Bluetooth. Next, the method should leverage pre-existing Bluetooth connection tools provided by the operating system. The biologists using this application are already familiar with these systems, so continuing to depend on them will allow the transition from Shoals to our new app to be more streamlined and seamless. Finally the communication method must be compatible with error detection and handling strategies. This is due to the imperfect nature of the wireless communication medium.

2.6.2: Alternatives

2.6.2.1: Direct Serial Port Profile Communication

This is the Classic Bluetooth approach. It emulates serial (COM) ports and opens virtual serial connections to exchange information with Bluetooth devices (in our case, the scanners). Connections use the RS-232 communication standard which allows for a consistent and standard data transfer. The app would leverage the OS's existing connection logic to allow the scanner to be connected. A socket-like connection would then be initiated in which the scanner will send information to the computer. This is a simple approach with widespread support in the form of libraries. Thanks to the use of virtual COM ports, this method also allows for higher throughput and an easily parsable datastream at the expense of increased power consumption and manual connection management.

2.6.2.2: Bluetooth Low Energy & GATT Services

This is the more modern Bluetooth approach. It is designed with low power consumption and low bandwidth communication in mind. The method is based on services and attributes that can be read from or written to via a wireless connection. While the power efficiency is a highly sought after quality, it is mitigated by the fact that most, if not all, of the scanners used by our client only support Bluetooth Classic by means of SPPs. This added with the fact that service handling logic can become complex puts BLE pretty low on the Bluetooth Communication Methods list.

2.6.2.3: Platform-Managed Connections

This approach lets the operating system take care of the complex connection logic while providing the app a simple interface by which to interact with the scanner. In this case users would simply connect the scanners to the laptop and the app would use the OS-provided API to interact with it. Having the OS take care of the majority of the complexity allows for a significantly reduced app-level complexity as it pertains to the communication with the scanner. The OS will handle all of the error checking, pairing UI, and reconnection at the expense of less control over timing and attributes like retries.

2.6.2.4: Managed Connection Layer

This approach consists of creating an entire connection management layer from scratch. This layer handles all of the reconnection, data buffering, and logging/recovery logic. The app would then set this module in charge of all of the communication with the scanner and bypass the OS entirely. While good for very specific applications where data needs to be consistently transmitted as well as for very specific transmission formats, this approach also adds high quantities of complexity to the project. Aspects like multithreading and error management are being done manually and can easily lead to bugs if not handled properly.

2.6.3: Analysis

The Bluetooth communication system is one of the more critical components of this application, as it directly interacts with the hardware that provides the efficiency the research teams are after. The system needs to be reliable, power efficient, and must correct any errors that occur during transmission.

2.6.3.1 Direct Serial Port Profile Communication

Serial Port Profile Communication is one of the two tried-and-true methods for wireless communication via Bluetooth. Because of this, there are many libraries and tools that aid in the connection establishment and management process. This was why testing bluetooth connectivity via this approach yielded good results. Despite not having one of the scanners used by the researchers in our possession, we were able to find Dart packages that helped access bluetooth devices. We were also able to create small Dart programs that could read the input of other Classic Bluetooth devices like remote controllers. Connection via this method does require a good amount of handling on the programmer's part, but is overall a stable and reliable approach.

2.6.3.2 Bluetooth Low Energy

The main difference between this method and Classic Bluetooth is that BLE has significantly reduced throughput and operates with a protocol that is not supported by a lot of devices. It's this latter difference that makes BLE unfeasible for this project. Our team found out pretty quickly that trying to connect to Classic Bluetooth devices using the BLE protocol resulted in no results at all. Connections could not be established and thus data could not be transferred. Judging as all of the bluetooth scanners used by our client use Bluetooth Classic, this method is unfeasible for this project.

2.6.3.3 Platform-managed Connections

This approach represents the extreme when it comes to complexity as it requires the developers to fully handle all aspects of the bluetooth connection for the specific platform they are working with. This made testing this method quite difficult as there were a lot of moving parts. Even despite being fully managed by the developer, the connection needed to go through the operating system, meaning doing all of that work was not entirely necessary when the operating system can handle most of it for you. Additionally, this connection logic would need to be implemented for both windows and linux. Overall, this level of control is much higher than is really required for our project, and while it can result in reliable connections and smooth interaction between the pc and the scanner, it simply isn’t worth the increased complexity and overhead that it generates.

2.6.3.4 Managed Connection Layer

On the other hand we have Managed Connection Layers which provide an interface with which to interact with the scanner. The users would connect the scanner to the computer and let the OS handle the connection, reconnections, and other logic. As developers, we would simply interact with the data that is returned from the scanner and not with the rest of the logic. This is, needless to say, a significantly simpler approach to interacting with the scanner, and similar tests to that of the Serial Port Profile tests show that connections can indeed be established and data read consistently. This combination of effectiveness and decreased complexity make this approach the one chosen by our team.

2.6.4: Bluetooth Communication With PIT Tag Reader Chart

	Direct SPP Communication	BLE	Platform Managed Connections	Managed Connection Layer
Support for Bluetooth Classic	✅	❌	🟡	✅
Leverages existing platform Bluetooth logic	🟡	🟡	🟡	✅
Supports Automated Signal Recovery	✅	✅	✅	✅
Can be used for error handling	✅	✅	✅	✅
Must work across a variety of firmware versions and scanners	✅	❌	🟡	✅

2.6.5: Chosen Approach

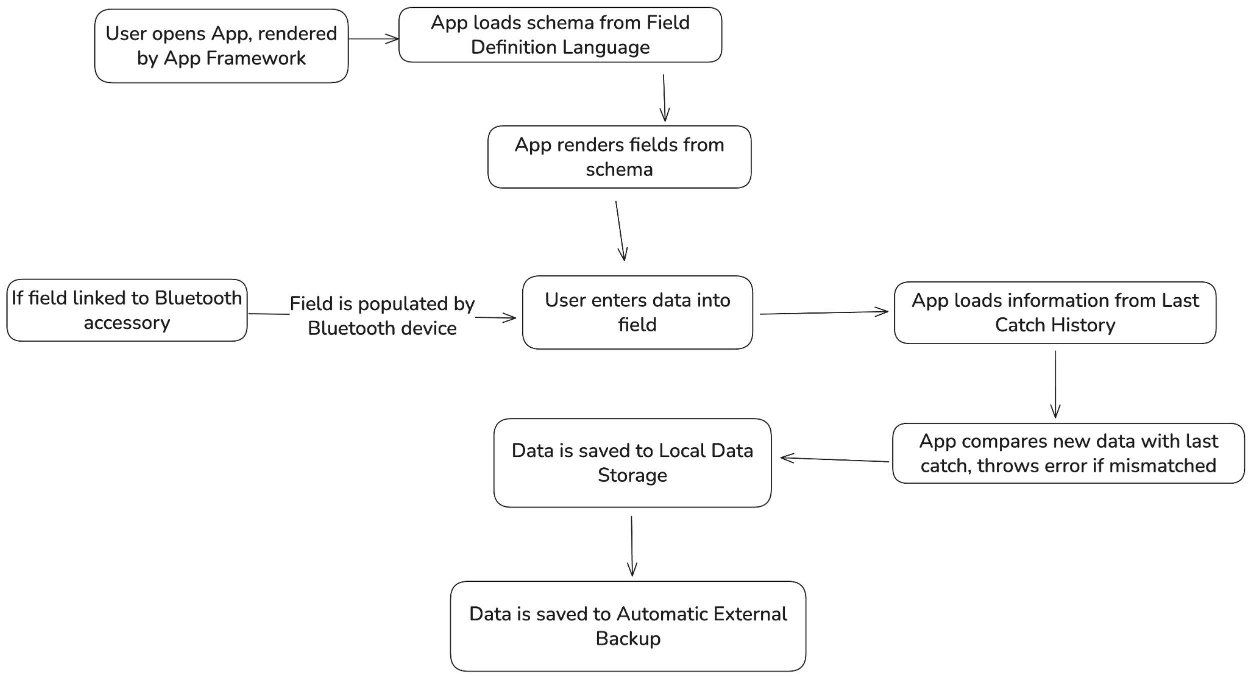
Since the project requires reliability in the field, and we want to avoid unnecessary complexity in the project, the best approach would be to leverage the tried and tested bluetooth logic provided by the OS and only handle the data being transmitted in our app. As seen in the above graphic, this method provides the best balance between reliability, ease of use, and simplicity while still being efficient.

2.6.6: Proving Feasibility

As briefly mentioned above, testing was conducted using a Classic Bluetooth controller. This was due to the lack of an actual scanner and PIT tags, though these should be in our possession in the near future. Although this testing was conducted in Dart, native helped functions were required to actually establish the connection. This is because native Bluetooth support is not yet available in Flutter for Classic Bluetooth devices. Despite this inconvenience a connection to the controller was indeed established and maintained, proving that connection is more than likely possible.

3.0: Technology Integration

Successful integration of these components is critical to delivering a dependable and user-friendly application. To achieve this, we must carefully consider how each element interacts within the system to create a unified, seamless experience. The following section provides a graphic representation of how we plan to integrate the various components of our application, along with a summary of the key integration details.


User opens App, rendered by App Framework
App loads schema from Field Definition Language
App renders fields from schema
If field linked to Bluetooth accessory
Field is populated by Bluetooth device
User enters data into field
App loads information from Last Catch History
App compares new data with last catch, throws error if mismatched
Data is saved to Local Data Storage
Data is saved to Automatic External Backup

The main component is our app framework and its underlying language, which most of the code will be written in. This will present the user interface and gather data from the user. If the field is configured to get data from a Bluetooth accessory, that data will be transferred to the app via Bluetooth and the app will populate the field with this data. This user interface will send data to the underlying language, which will then validate the data and attempt to store it in the local data storage, likely through a SQLite library. If the data is successfully saved to the local data storage, it will then be saved to the external backup.

4.0: Conclusion

The Colorado river is a crucial resource for the American southwest, with millions depending on it for their water needs. Having an understanding of the health of the ecosystem is therefore key to advising decisions which may impact the Colorado river. We are building a data entry app to aid field researchers in collecting crucial information for understanding the composition of the Colorado river’s various ecosystems. This document analyzes various technical challenges and weighs different possible solutions needed to build our application’s functionality. We analyzed different application frameworks, solutions for local data storage, and ways of adding Bluetooth functionality to our app. With this data, we may now make more informed decisions when designing our application’s architecture, which will result in a more polished final product. This application will help researchers effectively monitor the health and biodiversity of this ecosystem, helping to inform management decisions that will maintain the health of the Colorado river for years to come.