Event Driven Visualization Tool

JIB 4116

Arrington Goss, Daniel Han, David Martinez, Benjamin Skelton, Bharath Subu

Client: SLB Oilfield Services Company

GitHub Repository: https://github.com/Pineapple891/JIB-4116

Table of Contents

Table of Figures

Terminology

API (Application Programming Interface): A piece of software that acts as a “middleman” between two different software applications. The API defines and controls interactions and communications between two applications to allow the connected software components to work in sync.

Back-end: Refers to the parts of an application that the client is not expected to see or directly interact with. Back-end components often handle logic and other computations that the front-end needs to process user requests.

Cloud Storage: A method of storing data files in large “cloud servers” that are shared by several users. Files saved on a cloud server can be accessed remotely through any personal computer with internet access.

Embedded Device: A computer that is embedded within a larger device to perform computations and other processes that the device needs. Many objects that would not normally be considered a ‘computer’ make use of one or more embedded devices, including microwaves, thermostats, and pacemakers.

Firebase: A development platform owned and operated by Google LLC. Offers services to aid in online application development including authentication, NoSQL databases, cloud storage, and web hosting.

Front-end: Refers to the parts of an application that the client is expected to see and interact with. Front-end components include UI (user interface), user authentication, and input/output components.

JavaScript (JS): A programming language that is most often used for structuring the front-end of webpages and other applications.

NodeJS: A runtime environment that allows one to program the back-end logic of an online application using JavaScript code.

React: An open-source library that allows for creating front-end components (including user interfaces) using the JavaScript programming language.

React Native: A variant of the React library that is used to create applications that can run natively on various kinds of devices including mobile phones and personal computers.

Runtime Environment: A piece of software that provides the conditions to allow for other software modules to operate.

Version Control: The ability to view all previous versions of a file, compare differences between versions, and easily revert the state of a file to a previous version.

Visualization: Refers to any type of visual aid used to abstract information. In the context of this project, visualization refers to visual aids (including flowcharts) used to give a compact overview of one or more source code files.

Introduction

Background

When writing software for embedded devices, developers must create long-lasting code, as updating or maintaining the software can be challenging or even impossible, especially in special conditions like space. This makes it critical for engineers to thoroughly understand their code and how it affects the overall performance of the system throughout the device's lifespan. However, embedded systems' codebases are often highly complex, involving numerous interconnected events and components, making it difficult to grasp legacy code, particularly when documentation is limited.

At SLB, many developers currently rely on sub-optimal methods such as documentation analysis and manual testing to comprehend legacy codebases. Our group aims to address this issue by developing a visualization tool that offers a quick and simple way for SLB developers to analyze existing codebases. This tool will automatically generate easy-to-understand flowcharts from code, aiding in the understanding and documentation of system behavior. Additional features like version control, event-specific filtering, and interactive visualizations will further enhance the tool by allowing developers to observe code behavior in real time.

Document Summary:

The System Architecture section provides a high-level view of the components within our system and how they interact with each other during runtime. The system architecture is explained within this section in both a static and a dynamic context.

The Component Design section gives a detailed low-level description of our system and explains in more specific terms how our system's components operate with each other, and how these interactions are structured in both static and dynamic situations.

The Data Design section explains how our application stores and retrieves user data by making use of Google Firebase's cloud storage. This section also discusses how we plan on securing user data, and how we can provide secure methods for users to exchange files when using our application.

The UI Design section showcases the major UI screens of our application and explains how they function. This section also discusses the choices behind our choice of UI components and the design principles we took into account when making those decisions.

System Architecture

Introduction

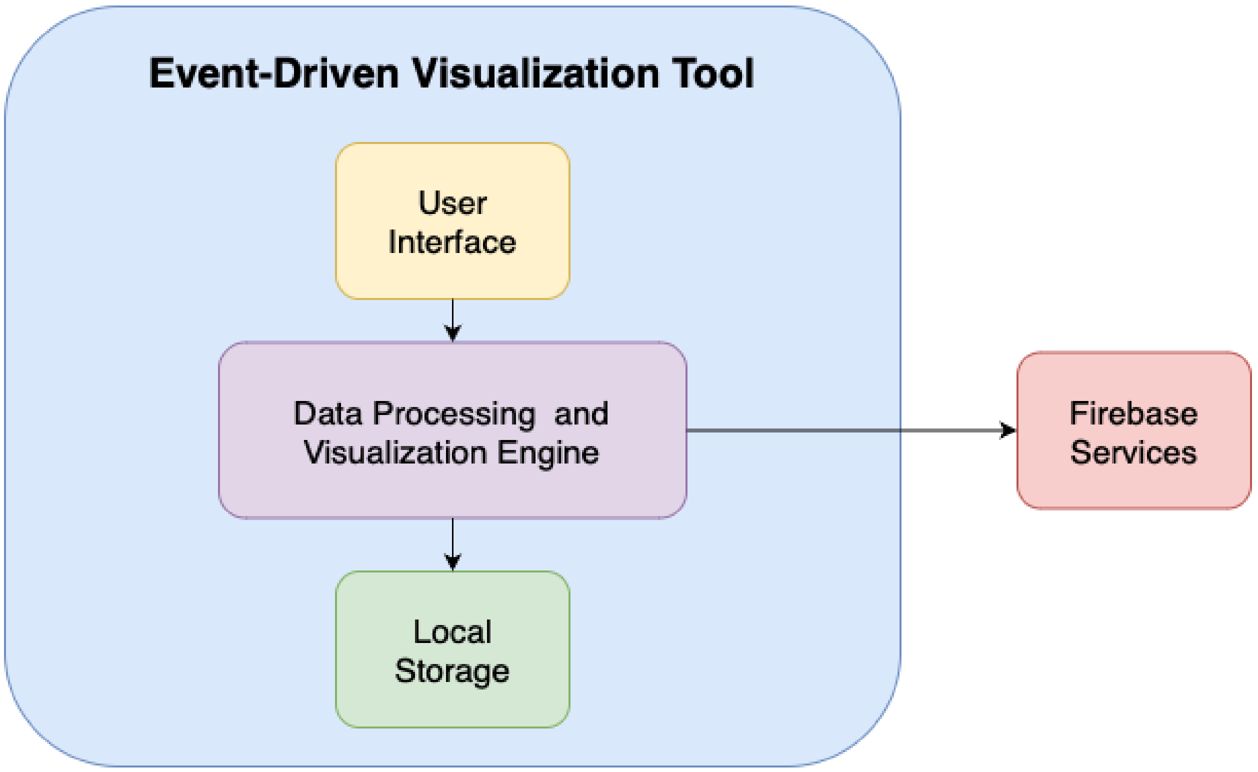
The system architecture of the Event-Driven Visualization Tool is designed to ensure provide a clear understanding of our system’s operation and structure. The static architecture diagram (Figure 1) highlights relationships between core components—User Interface, Data Processing Engine, Local Storage, and Firebase. The dynamic architecture diagram (Figure 2) illustrates real-time interactions during key operations, such as creating and saving visualizations, ensuring a seamless user experience. Together, these diagrams align with the tool's goals of reliable functionality, secure data handling, and intuitive workflows, while supporting future enhancements.

Rationale

Our static system architecture was designed in a way that highlights the core components of our system such as the User Interface, Data Processing and Visualization Engine, Local Storage, and Firebase are connected. It emphasizes foundational relationships between these components and displays an overview of the program's main components. The dynamic system architecture diagram focuses on how these components interact over time. Illustrating the sequence of events provides a user-centric perspective that explains system functionality. Both diagrams together provide a holistic view of our system's design.

Static System Architecture

The static system architecture diagram provides an overview of the system's core component, showing how each module, such as the User Interface, Data Processing and Visualization Engine, Local Storage, and Firebase Services, are interconnected. This diagram captures the relationships and dependencies between the components.


Event-Driven Visualization Tool
User Interface
Data Processing and Visualization Engine
Local Storage
Firebase Services

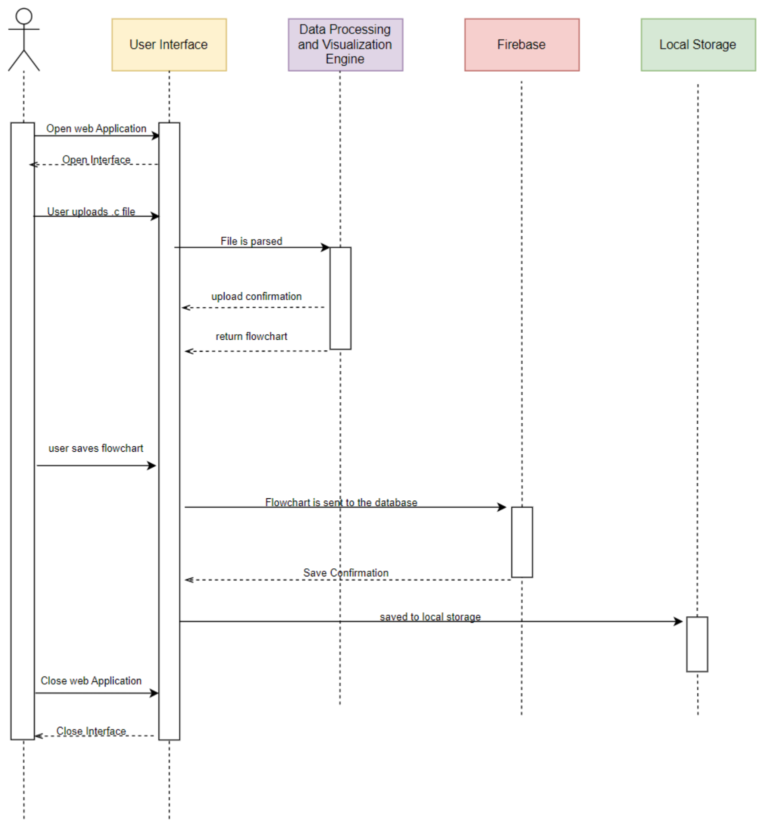
[Figure 1: Static System Architecture]

Description:

At the top, the User Interface module represents the front-end component of the application. This is where the users will interact with the system. This is then connected to the Data Processing and Visualization Engine, showing the flow of user commands to interact with the system. The Data Processing and Visualization engine will then create the visualization based of the user input and can then store the data into the backend Firebase Services or the Local Storage. The Firebase Services correlate to Authentication, Database, and Cloud Storage features.

Dynamic Architecture

The dynamic system architecture focuses on how components communicate to perform operations, showing their interactions over time in response to user inputs or events. We chose a sequence diagram (SSD) because it illustrates the flow of actions, providing a clear representation of system behavior during tasks like creating and saving visualizations. The SSD highlights how the User Interface sends commands to the Data Processing Engine, which generates visualizations and interacts with Firebase and Local Storage to store data.



[Figure 2: Dynamic System Architecture]

Description:

The Dynamic Diagram shows the interaction between the user and the application. It illustrates how all the components of the systems architecture interact with one another. In the diagram the scenario is that the user wants to create and save a visualization. The user will open the application and upload the c files. After the files are uploaded, the file is then parse and the engine will then create the visualization and allow the user to see it on the interface. Then the user requests to save the visualization to both the firebase services and to the local storage.

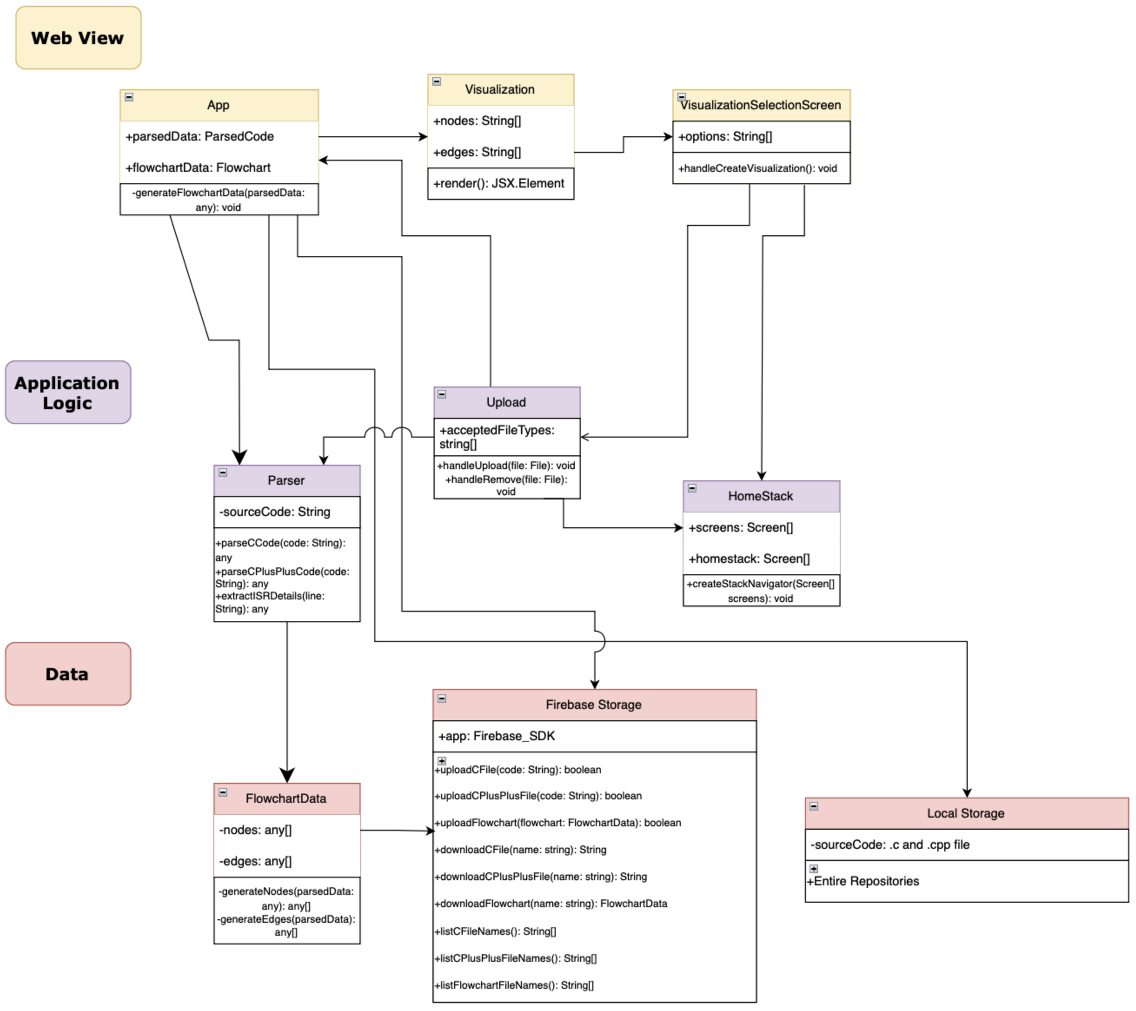
Component Design

Introduction

Below, we present two diagrams illustrating the high-level components that make up our app and their interactions. **Figure 3** is a static diagram that outlines each component's key data fields and methods, providing an overview of how data is structured within the system. This is essentially broken down into three components: the web view, application logic, and the backend data storage. This diagram also indicates the dependencies between components; for example, an arrow pointing from *Component A* to *Component B* signifies that *Component A* requires data from *Component B* to perform certain functions. This highlights how data flows and dependencies are managed within the app architecture.

Figure 4 shifts focus to the dynamic interactions between components, depicting the step-by-step process during a typical use case. In this scenario, a user uploads a C source code file to the tool, triggering the parsing component to analyze the code and generate a flowchart. The generated flowchart is then displayed in the user interface, allowing the user to interact with and view different parts of the code’s structure visually. Finally, the user can save the flowchart to Google Firebase Cloud Storage, which involves communication between the frontend components and the Firebase backend to ensure data persistence. This diagram emphasizes how user actions translate into interactions between components, demonstrating the seamless integration between the data processing, user interface, and cloud storage functions.

Static Elements

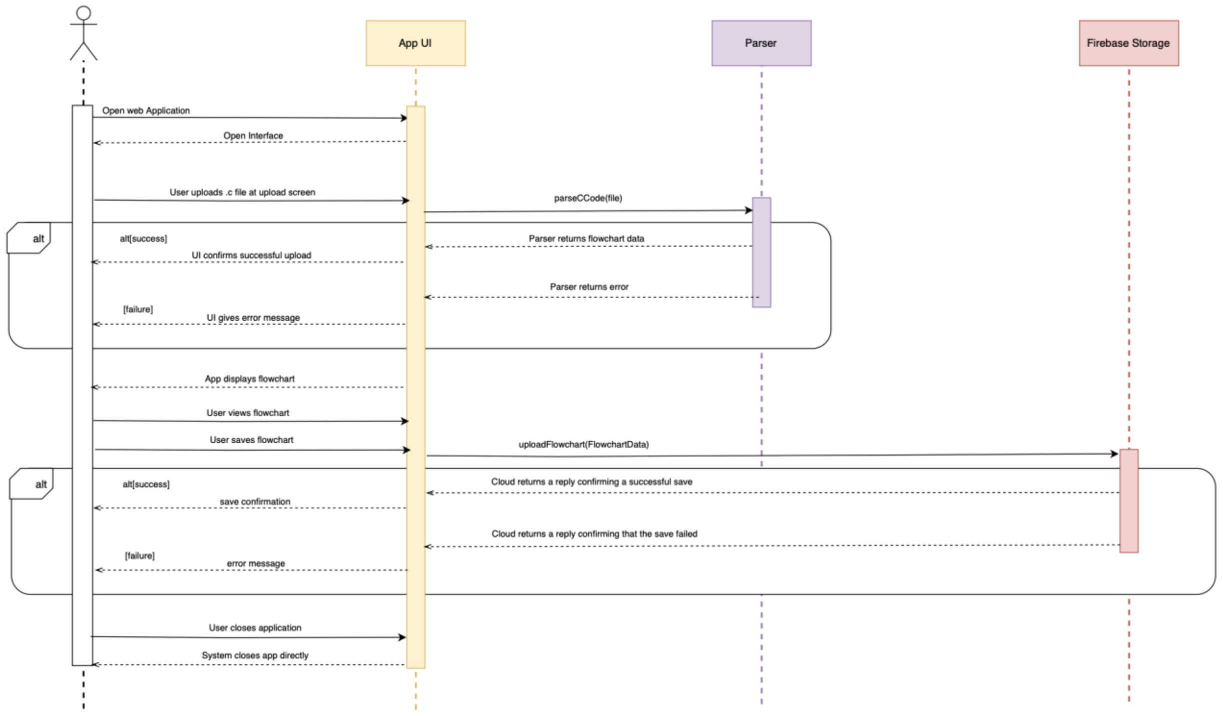

Web View
App
+parsedData: ParsedCode
+flowchartData: Flowchart
-generateFlowchartData(parsedData: any): void
Visualization
+nodes: String[]
+edges: String[]
+render(): JSX.Element
VisualizationSelectionScreen
+options: String[]
+handleCreateVisualization(): void
Application Logic
Upload
+acceptedFileTypes: string[]
+handleUpload(file: File): void
+handleRemove(file: File): void
Parser
-sourceCode: String
+parseCCode(code: String): any
+parseCPlusPlusCode(code: String): any
+extractISRDetails(line: String): any
HomeStack
+screens: Screen[]
+homestack: Screen[]
+createStackNavigator(Screen[] screens): void
Data
Firebase Storage
+app: Firebase_SDK
+uploadCFile(code: String): boolean
+uploadCPlusPlusFile(code: String): boolean
+uploadFlowchart(flowchart: FlowchartData): boolean
+downloadCFile(name: string): String
+downloadCPlusPlusFile(name: string): String
+downloadFlowchart(name: string): FlowchartData
+listCFileNames(): String[]
+listCPlusPlusFileNames(): String[]
+listFlowchartFileNames(): String[]
FlowchartData
-nodes: any[]
-edges: any[]
-generateNodes(parsedData: any): any[]
-generateEdges(parsedData): any[]
Local Storage
-sourceCode: .c and .cpp file
+Entire Repositories

[Figure 3: Static Component Design]

Description:

The component design diagram shows how different parts of the web application work together. The App is the main controller, managing data and interactions between other parts. The Parser processes the uploaded source code into usable data. FlowchartData creates the structure of the flowchart, which is then shown on the screen by the Visualization component. The Upload component handles file uploads, while Firebase Storage stores and retrieves files. The HomeStack manages navigation between different screens, and the VisualizationSelectionScreen lets users choose how they want to view the flowchart. This design helps keep the app organized and easy to manage.

Dynamic Elements



[Figure 4: Dynamic Component Design]

Description:

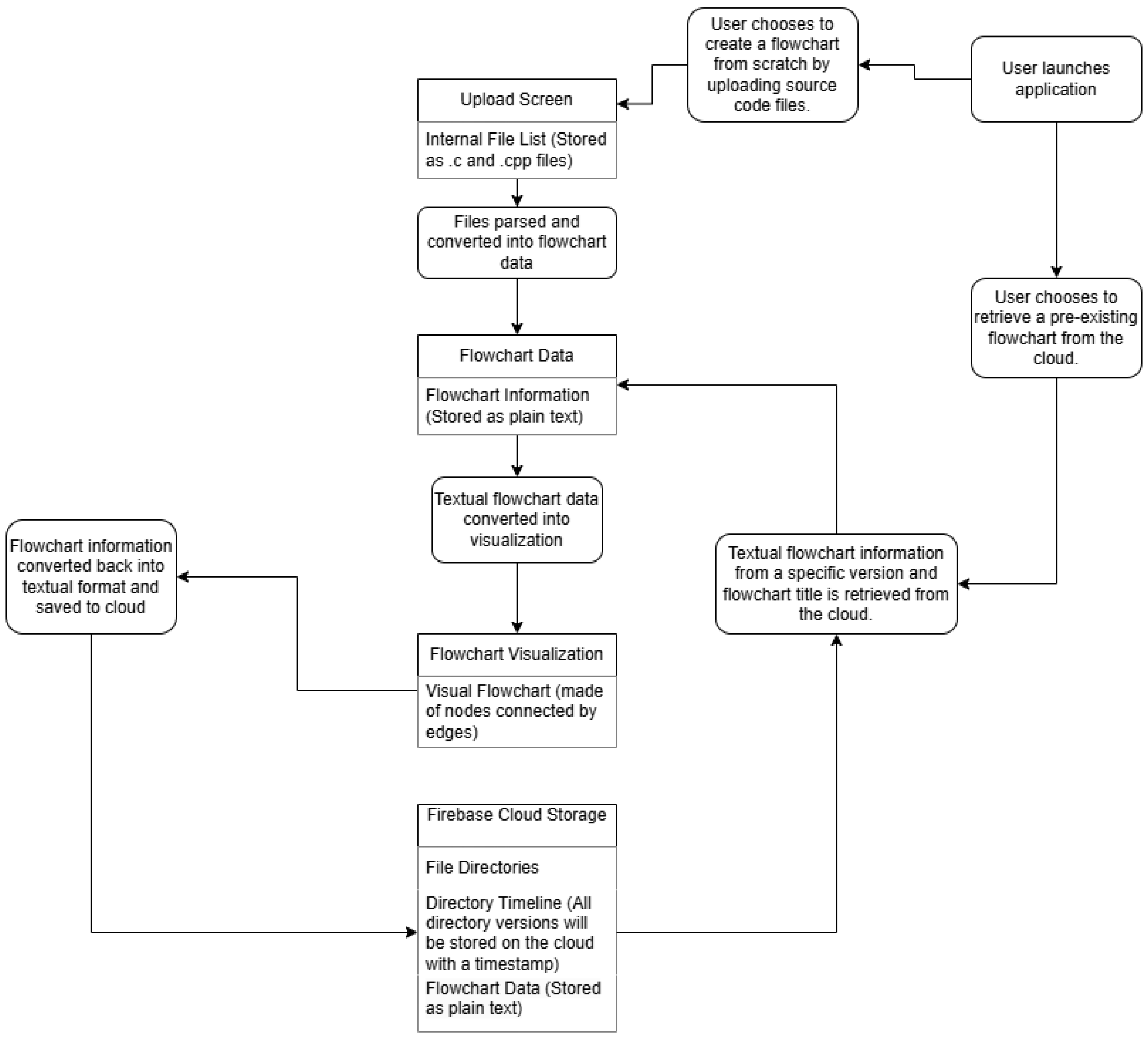
The Dynamic Component Design illustrates the workflow of a user interacting with a web application to upload a .c file, parse it into a flowchart, and save it to Firebase Storage. The process begins with the user opening the application and uploading the file through the App UI, which sends the file to the Parser for processing. If the parsing is successful, the Parser returns flowchart data, which the App UI displays for the user to view. The user can then save the flowchart, and the App UI sends the data to Firebase Storage. Firebase responds with either a success confirmation or an error message, which the App UI relays to the user. The diagram includes alternate flows for handling errors during both parsing and saving, ensuring robust feedback at every step of the interaction. Finally, the user can close the application, ending the session.

Data Design

Introduction

In this section, we will explore the data design considerations critical to the functionality of our system and how it interacts with Firebase Cloud Storage, a file-based NOSQL/non-relational database. This includes an overview of the database use diagram, which details the structure and interaction between the cloud database and user-generated data. We will also cover file use documentation, highlighting how user uploaded .c and .cpp files are parsed to generate the flowcharts. Additionally, we will discuss data exchange consideration, focusing on how our data is efficiently transferred between the user interface, parsing, and the cloud to ensure seamless flowchart generation and retrieval. These elements are necessary in maintaining data integrity and enhancing overall user experience.

Database Use Diagram


User chooses to create a flowchart from scratch by uploading source code files.
User launches application
Upload Screen
Internal File List (Stored as .c and .cpp files)
Files parsed and converted into flowchart data
Flowchart Data
Flowchart Information (Stored as plain text)
User chooses to retrieve a pre-existing flowchart from the cloud.
Textual flowchart data converted into visualization
Flowchart information converted back into textual format and saved to cloud
Textual flowchart information from a specific version and flowchart title is retrieved from the cloud.
Flowchart Visualization
Visual Flowchart (made of nodes connected by edges)
Firebase Cloud Storage
File Directories
Directory Timeline (All directory versions will be stored on the cloud with a timestamp)
Flowchart Data (Stored as plain text)

[Figure 5: Database Use]

Description:

All of the data in use by our application can be separated into one of three types, Raw Source Code Files, Flowchart Information stored as text, and the data that makes up the visual flowchart that is presented to the user.

When a user starts up our application, they have a choice of either creating a flowchart from scratch or downloading a pre-existing flowchart from the cloud. If the user chooses to create a new flowchart, then they will proceed to the upload screen to upload one or more source code files (which must be in either of the types .c or .cpp) to the application. These source code files are then run through a parser which creates a textual representation of a flowchart. The user then proceeds from the upload screen to the flowchart screen, where the textual flowchart information is converted into a visual representation made up of nodes containing textual information; related nodes are connected by an edge. On the flowchart screen, the user can choose to save the flowchart to the cloud, which results in the visual flowchart being converted back into a textual format which is then saved to cloud storage.

To allow for version control, each flowchart saved on the cloud is stored in a directory corresponding to the name the user saved the flowchart under. Flowchart versions saved under the same name are stored in the same directory, with each version being marked with a time stamp (marking the date and time that version was saved to the cloud) to differentiate them from each other.

A user who chooses to download a pre-existing flowchart from the cloud will be brought to a separate screen where they will be prompted to select from one of the flowchart directories that exist in cloud storage. After selecting a directory, the user is then presented with the list of versions that exist in the directory. Once the user chooses a version, that version's textual flowchart information is downloaded to the application, and the user can proceed to the flowchart screen.

File Use Documentation

Our program interacts with a variety of files, primarily focusing on .c and .cpp files. Our users would upload entire repositories containing them, which the system then parses to extract relevant code for the flowchart. There are no specific formatting requirements. The program is designed to handle various coding styles. The information parsed from these files is used to generate a flowchart that visually represents the code's logic and structure. Users have the option to save these flowcharts to a cloud storage system in order to access them in the future. To make storing flowcharts simpler, they may be converted into a text file before being saved on the cloud. These text files will contain all the flowchart's information, including lists of nodes and links present in the chart.

Data Exchange Considerations

Our flowchart tool involves data exchange between the user interface and the backend Firebase Cloud Storage system. The data being transferred includes the uploading and downloading of flowchart data in a text-based format. The data formatting involves standard file types for code files (.c and .cpp) and JSON structures for storing and retrieving flowchart data from the Firebase database.

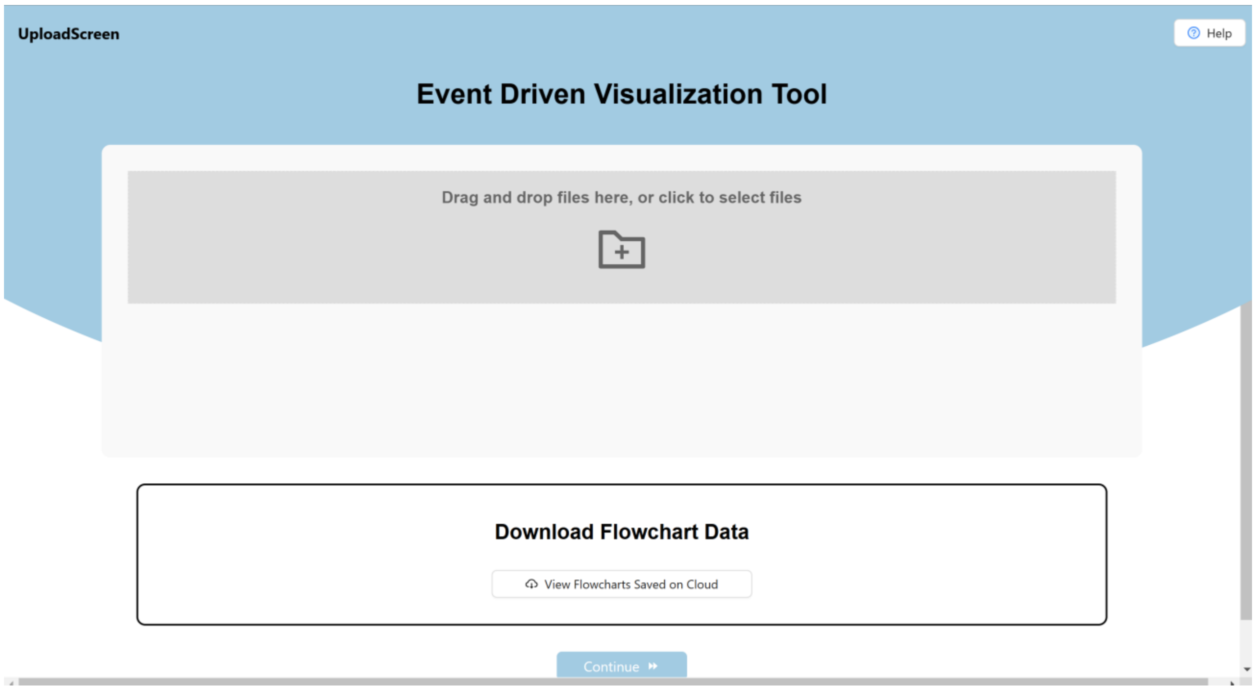
Our client expects to be able to use this tool with their own codebases. There is a moderate risk of damage to the company if source code files or flowchart files are stolen during data transit. While we do not expect for any of these source code files to contain a large amount of PII (Personally Identifiable Information), we understand that these source code files are considered the company’s property and must be protected. Currently, we are focusing on implementing the functionality of the app with little priority for security concerns, so we have not added any security measures to our tool. However, there are several areas where we can make improvements in security. These include encryption of HTTPS and User Authentication.

Firebase, the platform we are using to host our cloud storage, already encrypts uploads and downloads using HTTPS, in addition to encrypting data saved to the cloud. In addition, firebase cloud storage can have security rules implemented, which can govern what kinds of actions different users can perform with the database. For one example, we could enforce a security rule to only allow users to save text documents that are below a certain file size, which would make it harder for an adversary to fill up the available storage with extraneously large files. Overall, while firebase may not be the most secure option for cloud storage, we believe that the platform can provide us with a level of security that is appropriate for the types of files this application will use.

UI Design

For our overall interface, we chose to use a simple design that focuses on the needs of our users. Our application currently consists of three main pages. The application launches in the Upload Screen, where a user can upload their directory and view files in our preview pane. After clicking continue, the user is then sent to the Flowchart Visualization, where they can interact and edit their flowchart as they please.

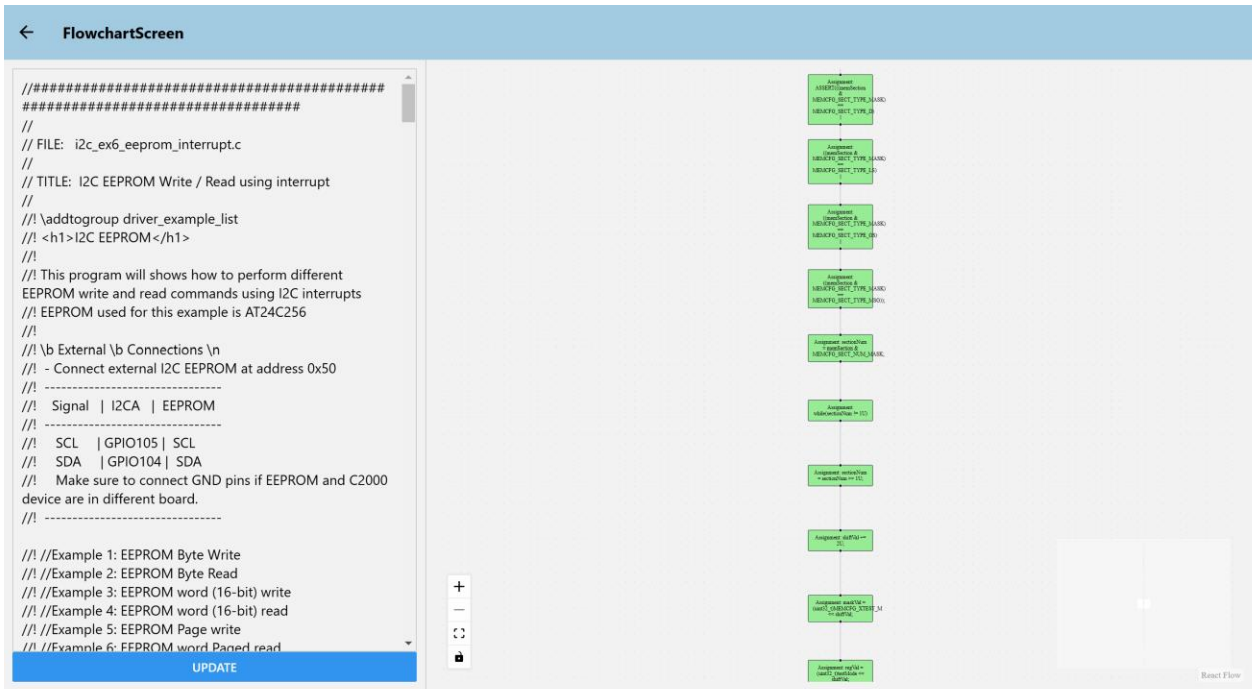
Below is our upload screen after a user has uploaded a directory. The file list below the grey upload zone serves as a visual confirmation that the user has uploaded the correct files. We've also included the classic "Drag and Drop" phrase so that the interface feels more comfortable for our users.


UploadScreen
Help
Event Driven Visualization Tool
Drag and drop files here, or click to select files
Download Flowchart Data
View Flowcharts Saved on Cloud
Continue

[Figure 6: Upload Screen]

Description:

Users can upload source code files on this screen. Uploaded files are displayed in a scrollable list to allow users to confirm that the upload was successful. This visual confirmation aligns with the Visibility of System Status Nielson heuristic.


FlowchartScreen
UPDATE
Assignment: ASSERT(((memSection & MEMCFG_SECT_TYPE_MASK) == MEMCFG_SECT_TYPE_D)
Assignment: ((memSection & MEMCFG_SECT_TYPE_MASK) == MEMCFG_SECT_TYPE_LS)
Assignment: ((memSection & MEMCFG_SECT_TYPE_MASK) == MEMCFG_SECT_TYPE_GS)
Assignment: ((memSection & MEMCFG_SECT_TYPE_MASK) == MEMCFG_SECT_TYPE_MSG));
Assignment: sectionNum = memSection & MEMCFG_SECT_NUM_MASK;
Assignment: while(sectionNum != 1U)
Assignment: sectionNum = sectionNum >> 1U;
Assignment: shiftVal += 2U;
Assignment: maskVal = (uint32_t)MEMCFG_XTEST_M << shiftVal;
Assignment: regVal = (uint32_t)testMode << shiftVal;
React Flow

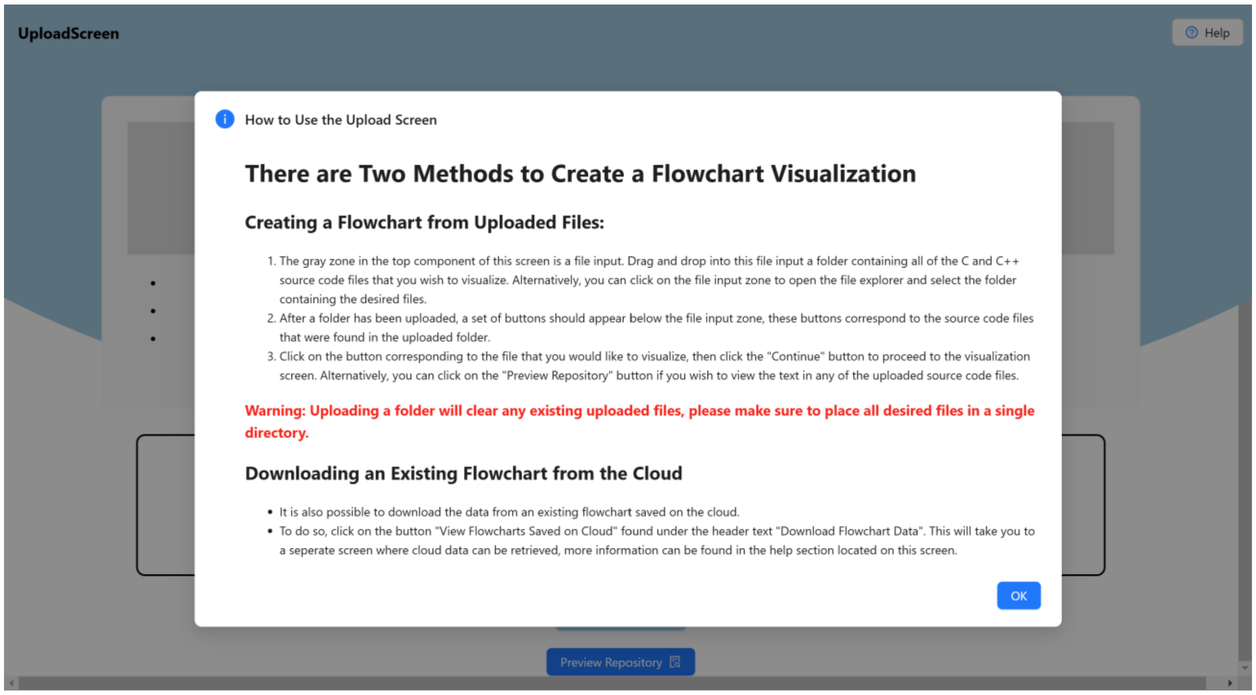
```
//#############################################
###################################
//
// FILE:   i2c_ex6_eeprom_interrupt.c
//
// TITLE:  I2C EEPROM Write / Read using interrupt
//
//! \addtogroup driver_example_list
//! <h1>I2C EEPROM</h1>
//!
//! This program will shows how to perform different
EEPROM write and read commands using I2C interrupts
//! EEPROM used for this example is AT24C256
//!
//! \b External \b Connections \n
//!  - Connect external I2C EEPROM at address 0x50
//! --------------------------------
//!    Signal   |  I2CA   |  EEPROM
//! --------------------------------
//!     SCL     | GPIO105 |  SCL
//!     SDA     | GPIO104 |  SDA
//!    Make sure to connect GND pins if EEPROM and C2000
device are in different board.
//! --------------------------------

//! //Example 1: EEPROM Byte Write
//! //Example 2: EEPROM Byte Read
//! //Example 3: EEPROM word (16-bit) write
//! //Example 4: EEPROM word (16-bit) read
//! //Example 5: EEPROM Page write
//! //Example 6: EEPROM word Paged read
```

[Figure 7: Flowchart Visualization Screen]

Description:

After completing the steps for generating a flowchart, the user will then be taken to the Flowchart Visualization Screen where they can view and interact with a flowchart showing their data. Users are also able to go back to the upload screen from this page, aligning with Nielson’s User Control and Freedom Heuristics.



How to Use the Upload Screen

There are Two Methods to Create a Flowchart Visualization

Creating a Flowchart from Uploaded Files:

1. The gray zone in the top component of this screen is a file input. Drag and drop into this file input a folder containing all of the C and C++ source code files that you wish to visualize. Alternatively, you can click on the file input zone to open the file explorer and select the folder containing the desired files.
2. After a folder has been uploaded, a set of buttons should appear below the file input zone, these buttons correspond to the source code files that were found in the uploaded folder.
3. Click on the button corresponding to the file that you would like to visualize, then click the "Continue" button to proceed to the visualization screen. Alternatively, you can click on the "Preview Repository" button if you wish to view the text in any of the uploaded source code files.

Warning: Uploading a folder will clear any existing uploaded files, please make sure to place all desired files in a single directory.

Downloading an Existing Flowchart from the Cloud

- It is also possible to download the data from an existing flowchart saved on the cloud.
- To do so, click on the button "View Flowcharts Saved on Cloud" found under the header text "Download Flowchart Data". This will take you to a seperate screen where cloud data can be retrieved, more information can be found in the help section located on this screen.

[Figure 8: Help Screen]

Description:

On each page of our program, we have included a help button that leads to a detailed guide to using our program that aligns with Nielson's accessibility heuristic.

Appendix

Google Firebase Cloud Storage API

Documentation URL

https://firebase.google.com/docs/storage

API Information

Input/Response Format: Stringified JSON Object

Requires Authentication: No, but authentication can be set up for Firebase Cloud Storage

Rate/Storage Limited: Without a paid subscription to Google's Blaze Plan, the maximum allowed data transfer rate is 1 GB per Day, and up to 5 GB of data can be stored on the cloud at any given time.

Example Request

```
const downloadReference = ref(storage, 'flowcharts/example.txt');

const downloadedFlowchart = getBytes(downloadReference);
```

Example Response

```
{"nodes":[{"id":"flowchart-
0","type":"Process","data":{"label":"Operation or
Assignment"},"position":{"x":100,"y":50}},{"id":"flowchart-
1","type":"Process","data":{"label":"Operation or
Assignment"},"position":{"x":100,"y":200}},{"id":"flowchart-
2","type":"Process","data":{"label":"Operation or
Assignment"},"position":{"x":100,"y":350}},{"id":"flowchart-
3","type":"Process","data":{"label":"Operation or
Assignment"},"position":{"x":100,"y":500}}],"edges":[{"id":"edge-
flowchart-0-1","source":"flowchart-0","target":"flowchart-
1","animated":true},{"id":"edge-flowchart-1-2","source":"flowchart-
1","target":"flowchart-2","animated":true},{"id":"edge-flowchart-2-
3","source":"flowchart-2","target":"flowchart-3","animated":true}]}
```