PharmaSUG China 2023 - Paper AA-127

A Socket-Based Collaborative SAS Editor

Furan Wang, Pfizer (China) Research & Development

ABSTRACT

This paper presents a collaborative solution that enables multiple users on different PCs to develop SAS code together. The approach, implemented using Python3 and JavaScript under the Flask framework, incorporates a code editor with comprehensive Dynamic Syntax Highlighting. The fundamental technique behind this solution is to apply the 'web socket' protocol, which facilitates full-duplex communication channels over a single TCP connection. In comparison to traditional SAS code editing, the collaborative programming approach offers numerous advantages, including simplified problem-solving, enhanced communication, and improved mentoring efficiency. The editor also includes interfaces for GPT plugin, allow our sas programmers asking GPT to collaborate on generating code snippets, identifying programming errors, and receiving suggestions for improving the code logic in the form of a single question with zero data submissions, within a certain range of data security.

INTRODUCTION

In today's technology-driven world, collaboration is a key factor in achieving success. Particularly in software development and data analysis, team members may need to work together to tackle complex problems and develop innovative solutions. However, collaborative work can be challenged by geographical limitations, time differences, and information sharing barriers.

To overcome these obstacles and improve team efficiency, a bunch of companies release their platform products which support these requests, such as Live Share, CodePen , GItLab, and so on. But there is no production especially for SAS. This may be because most of SAS programmers work in industries such as healthcare, biochemistry, and clinical fields, where there are requirements for the programmers to adhere to a code responsibility system. While there is still some application scope within teaching and debugging, the demand for collaborative SAS development is not urgent overall. However, with the advent of GPT technology [Brockman et al. (2016)], we could now invite GPT to participate in collaborative development as a team member, which will enhance the experience of writing SAS programs significantly. So that is why we believe it is the right time to have a proper collabrative editor for SAS. Since WebSocket technology was initially proposed by Ian Hickson in 2008 and standardized as RFC 6455 in 2011 [Hickson (2011)]. It establishes a persistent connection channel between the client and the server, enabling low-latency and efficient realtime communication. Over the past decade, WebSocket technology has made rapid progress. With its advantages of real-time communication [Pimentel & Nickerson(2012)], low latency [Pimentel & Nickerson (2012)], bidirectional communication [Lombardi (2015)], and high scalability [Perez et al. (2020)], it has found extensive applications [Muller (2014)] in the era of HTML5. Compared to the traditional HTTP request-response model, WebSocket eliminates the overhead of multiple handshakes and header information, reducing network traffic and latency.

Based on on these prerequisites and advanced technologies, a socket-based collaborative SAS editor has been introduced. The intention is to help SAS programmers achieve higher efficiency, greater flexibility, and more convenient programming experience, thereby delivering superior submissions.

WORKING PRINCIPLES

The socket-based collaborative SAS editor allows multiple team members to simultaneously edit and view the same SAS code or analysis scripts. By utilizing network socket connections, the editor enables real-time code synchronization and team collaboration. When one team member makes changes to the code, those changes are immediately reflected in the editor windows of all other members. This real-time synchronization capability enables team members to collaborate more quickly and efficiently, eliminating the hassle of traditional file transfers or version control systems.

TECHNIQUE OUTLINES

The implementation of a Socket-Based Collaborative SAS Editor using Flask and JavaScript involves the following technical details:

1. **Server-Side (Flask) Implementation**:

- Create a Flask application and initialize the Flask-SocketIO extension.

```
from flask import Flask, render_template
from flask_socketio import SocketIO
app = Flask(__name__) #Create Flask application
socketio=SocketIO(app) #Initialize SocketIO
...
```

Code 1: initializing

- Define routes to handle client requests and return the appropriate pages or data.

```
#Define a route to handle the root URL
@app.route('/')
def rootUrl():
    return render_template('Main.html')

#Define a route for a message diliver
@app.route('/api/<data>')
def get_data(data):
      data = {'message':data}
      return jsonify(data)
...
```

Code 2: routes defining

- Use the @socketio.on decorator to define SocketIO event handlers for processing events sent by clients, such as connecting, disconnecting, and editor content updates. Within the event handlers, use the emit function to send events and data back to clients or use the broadcast=True option to broadcast data to all connected clients.

```
#Host a new channel for code editting.
@socketio.on("new channel created")
def create_new_channel(data):
      channel = clean_up_channel_name(data["channel"])
      app.logger.info( "HINT: Running the 'create new channel'
      "f"function with channel: {channel}.")
      # If the channel is new, add it
      if channel not in channels:
            if valid_channel(channel):
                  channels.append(channel)
                  # Add the channel item to the messages dicationary
                  messages_dict[channel]=[]
                  app.logger.info(f"New Channel creation success with
                  \n\n{channels}!""\n\ nEmitting 'add channel'\n\n")
                  emit("add channel",{"channel": channel}, broadcast=True)
            else:
                  # If the channel is NOT a valid name
                  emit("invalid channel name", broadcast=False)
      else:
            app.logger.info("The Channel is in using!")

            emit("channel creation failed",broadcast=False)

#Processes the new line of SAS script and stores it into the server
@socketio.on("edit code")
def edit_code(data):
      ...
      data = {
            "sascode": sascode,
            "user": user,
            "timing":time,
            "CodeLoc":CodeLoc,
            "Editors":EdiLis
      }
      emit("code receive", data, broadcast=True)

#Process the "Ask "Join event
@socketio.on("Ask Join")
def ask_join(data):
      ...

      emit("Request Receive", data, broadcast=True)

#Process the "Accept "Join event
@socketio.on("Accept Join")
def request_accept(data):
      ...
      emit("Request Accept", data, broadcast=True)
```

Code 3: SocketIO events

2. **Client-Side (JavaScript) Implementation**:

- Import the Socket.IO client library in the HTML page to establish a WebSocket connection with the server.

```
//Include the Socket.IO client library in your HTML
   file by adding the following <script> tag to the <
   head> section:
<script src="../socket.io/4.4.1/socket.io.js"></script>
```

Code 4: socket.io.js Importing

- Create a Socket.IO client instance, connect it to the server using the io.connect() function and initialize it.

```
// Create a Socket.IO client instance based on current
   address.
    const socket = io.connect(
        `${window.location.protocol}//${document.domain
          }:${window.location.port}`
    );

// Initializing socket by the a "handshaking" with the
    server
      socket.on('Initializing', () => {
            ...
            return false;
    };

//  Initializing  connection_error  modulor
    socket.on('connect_error', (error) => {
    ...
    return false;
    });
```

Code 5: Client-side Socket.IO Initializing

- Use event listeners provided by the Socket.IO client, such as socket.on(), to listen for events and data sent by the server. At appropriate points, such as when the editor content changes, or clients hosting-joining events, use the socket.emit() function of the Socket.IO client to send data to the server.

```
//Estabish the connection between Server-Side and Client-Side.
socket.on('connect', () => {
//the editor content changes
   $("#sascode").on("keyup",
   function(e){
   ...
      socket.emit('editcode',
      { sascode: sascode,
      user: USER,
      timing: time().toString(),
       CodeLoc: CodeLoc
      });
   };
}

//Client ask to join var state = $('#STATE').text();
if (state.includes('Ask Join')) {
var Jointer = $('#PersonID').attr('title');
var Admin = $('#Admin').text(); var socket = io();
socket.emit('Ask Join', {
     Requester: Joiner,
     Admin: Admin
   }); return
   false;
}

//Channel Owner Recieve Request
socket.on('Request Receive', data => {
   var USER = $('#PersonID').attr('title');
   var Admin=data['Admin'];
   if (USER==Admin){
   var Requester=data['Requester']
   if (confirm(Requester+' wants to join this
            session. Accept or not?')){
           socket.emit('Accept Join', {
                Requester: Requester,
                Admin: Admin,
                Host: $('#HOST').val();,
                SftpLoc: $('#CodeLoc').val();
                });
        Return false;

        }else{ socket.emit('Reject Join', {
                user: Requester, Admin:
                Admin
                }); return
          false;
        }
   } };
});
...
```

Code 6: client-side listener

3. **Editor Implementation**:

- Add a ”code” element in the HTML page for editing SAS code with ”contenteditable” value true, and giving it a unique identifier, id=”sascode”.

```
<code class="SAS"
id="sascode"
contenteditable=true
style=
 "display: inline-block;left: 3.5%;
   white-space:pre-wrap;
   overflow-y:scroll;
   outline: dotted thingrey;
   outline: none;
   border: none;
   margin: 0 0 5px;
   padding: 5px;width: 96%;
   position: absolute;
   height: 88%;
   background: #fff;
   overflow: auto;
   font-size: xlarge;"
   spellcheck="false">
```

Code 7: SAS Editor

- Use Jquery to retrieve the editor element ”sascode” and listen for input events.

```
// Always prefer the 'keyup' event instead of 'input'
$("#sascode").on("keyup", function(e)
```

Code 8: key pressed listener

- Utilize highlight.js to apply syntax highlighting to the script once the user presses any key.

```
...

function highlightSyntax()
{ ...

  var code = $('#sascode').text(); ...

 //get Cursor position.
  var savedSel = rangy.saveSelection();

  var CodeWithTags=$("#sascode").get(0).innerHTML;

  var keyword='selectionBoundary';

   var tempLis=CodeWithTags.split(keyword)[0].split('<
     span');
   var CodeB4kw=tempLis[tempLis.length-1];

  var CodeAFkw=CodeWithTags.split(keyword)[1].split("
 span>")[0];

  Var rangyFullTag='<span'+CodeB4kw+keyword+CodeAFkw+
        'span>';
     //replace the cursor within a <span>
     ...
     //highlight the SAS script.
 Var ins= $("#sascode").get(0).innerHTML.split( keyword);

     $("#sascode").get(0).innerHTML=CodeWithoutTags.
        replaceAll('\n\n','\n').replace('&#xfeff','');

     hljs.highlightBlock($("#sascode").get(0));
     ...
}
...
```

Code 9: Dynamical Syntax highlighting

- When the editor content changes, use the socket.emit() function of the Socket.IO client to send the updated content to the server.

```
...
//get each line of code
Var tempLis=$("#sascode").get(0).innerText.trim()
.split('\n');
//trim each line

for (var i in tempLis)
  {
   tempLis[i]=tempLis[i].trim();
   }

Var CodeLoc=$("#CodeLoc").get(0).value;
var sascode=tempLis.join('\n');
var USER = $('#PersonID').attr('title');
$("#sascode").get(0).focus();
socket.emit('edit code',
    { sascode: sascode,
        user: NTID,
        timing: time().toString(),
        CodeLoc: CodeID
    }
);
...
```

Code 10: client-side socket emitting

• Upon receiving the editor update event sent by the server via Socket.IO client, update the content of the editor.

```
...
//Code receive event listener
socket.on('code receive', data => {
var sascode=data['sascode'];
var Editor=data['user'];
var Moditime= data['timing'];
var USER=$("#PersonID").get(0).title;
var CurCodeLoc=$("#CodeLoc").get(0).value;
var CodeLoc=data['CodeLoc'];
      //Check whether current code is edited by current user
      if ((Editor!=USER) && (CurCodeLoc==CodeLoc)) { let savedSel
            = rangy.saveSelection();
            var CodeWithTags = $("#sascode").get(0).
                innerHTML;
            var keyword = 'selectionBoundary'; //Check whether user
            is currenting editing
            if (CodeWithTags.indexOf(keyword)>0) { ...
                  //Get current USER cursor postion
                  var tempLis = CodeWithTags.split(keyword)
                      [0].split('<span')
                      var CodeB4kw = tempLis[tempLis.length - 1];
                      var CodeAFkw = CodeWithTags.split(keyword)
                      [1].split("span>")[0];
                      var rangyFullTag = '<span' + CodeB4kw + keyword
                  + CodeAFkw + 'span>'; ...
                  //Replace USER cursor with a special string tag
                        '@CURSOR@'
                  $("#sascode").get(0).innerHTML=CodeWithTags.replace
                      (rangyFullTag,  '  @CURSOR@').replaceAll('<br>',
                      ''). replaceAll('\n\n', '\n');//clearfy current
                      SAS code
                  $("#sascode").get(0).innerHTML=$("#
                      sascode").get(0).innerText.replaceAll('\n\n',
                      '\n');//.replaceAll ('/<[^/>][^>]*><\/[^>]+>/gim','');
                  //get current SAS code
                  var cursascode = $("#sascode").get(0).
                  innerText.trim();
                  //apply self-developed function 'htmldiff' with current
                      sas code and received code
                  let output = htmldiff(cursascode, sascode);
                  $("#sascode").get(0).innerHTML = output;
                  hljs.highlightBlock($("#sascode").get(0));
                  //restore USER's cursor
                  rangy.restoreSelection(savedSel);
                       $("#sascode").get(0).focus();
            }else{ var cursascode = $("#sascode").get(0).
                        innerText.trim();
                  let output = htmldiff(cursascode, sascode);
                  $("#sascode").get(0).innerHTML=output;
                  hljs.highlightBlock($("#sascode").get(0));
            }
            return false;
      }
 });
...
```

Code 11: receive updated code from Server-side

4. **Data Persistence**:
   - To ensure data persistence, we use 'flask_sqlalchemy' to store and retrieve SAS code from the server.

```
app.config['SQLALCHEMY_DATABASE_URI'] = \
 'sqlite:/// database.db'
db = SQLAlchemy(app)
```

Code 12: DataBase Construction

   - When clients connect or make changes, the server can update the corresponding data in the database to maintain the state of the collaborative SAS editor.

```
...
#Define a data model class that represents the data to
   be stored in the database.
class EditorData(db.Model):
      id= db.Column(db.Integer, primary_key=True)
      sascode = db.Column(db.Text)
      timestamp = db.Column(db.DateTime)
      ...
```

Code 13: DataBase Initializing

```
 ...
@app.route('/connect')
def connect():
     # Create a new EditorData record for the connected client
     snippet = EditorData(sascode='', timestamp=datetime
        .datetime.now())
     db.session.add(snippet)
     db.session.commit()
     # Return the ID of the new record to the client
     return str(snippet.id)
@app.route('/update/<int:snippet_id>', methods=['POST'])
 def update(snippet_id):
     # Retrieve the EditorData record based on the client's ID
     snippet = EditorData.query.get(snippet_id)
     # Update the sas edit field with the new code
     snippet.sascode = request.form['sascode']
     snippet.timestamp = datetime.datetime.now()
     # Commit the changes to the database
     db.session.commit()
 ...
```

Code 14: DataBase operation

5. Error Handling and Exceptional Cases:

- We design various error scenarios, such as network disconnections or server crashes, and implement appropriate error handling mechanisms on the serverside and client-side to provide a robust and reliable application.

```
@socketio.on('connect_error') # deal with connect_error
def handle_connect_error(e):
   Report("Connect Error:", e)
   error_message=str(e)
   ...
   emit('error', error_message)
@socketio.on('disconnect') # deal with disconnect def
handle_disconnect():
error_message = 'Disconnected from the server'
     ...
     emit('error', error_message)
     # send error messages to client-side
 ...
```

Code 15: Connection error sending back

```
# define a dictionary to store connected clients and their sas
   code states
connected_clients ={}
@socketio.on('connect')
def handle_connect():
     client_id = request.sid
          connected_clients[client_id] = {'sascode': ''}

@socketio.on('disconnected')
def handle_disconnected():
       client_id = request.sid
       if client_id in connected_clients:
             del connected_clients[client_id]
       disconnect()

@socketio.on('update_sascode')
 def handle_update_sascode(data):
     client_id = request.sid
     if client_id in connected_clients:
          # Update the sascode for the connected client
          connected_clients[client_id]['sascode']=data['sascode']
          # Broadcast the updated sascode to other clients
          socketio.emit('sascode_updated',
          {'client_id':client_id,
          'sascode': data['sascode']},
          broadcast=True)
     else:
          disconnect()
          ...
```

Code 16: Connection events handling

INTEGRATION WITH GPT API

The integration process can be summarized by following these steps:
1.Obtain API credentials: Sign up for an API key from OpenAI and ensure the necessary credentials to access the GPT API.
2.Set up the API client: Install the required Python libraries and set up the API client in our server-site environment.
3.Develop the logic for prompt: Determine a class of GPT which handles the prompts to provide corresponding action with the GPT model.
4.Make API requests: Use the API client to send requests to the GPT API. Pass users' prompt as input to the API, specifying the desired model as GPT3.5 and attaching any additional parameters users customized.
5.Process the API response: Receive the response from the GPT API, which will contain the generated SAS code or suggestions for users programming task.
6.Integrate with front-side: Extract the relevant information from the response and render it to UI.

A concise outline of the technique behind is provided below:

- To use OpenAI in Python, we start with installation of the OpenAI Python package and obtain valid API credentials.

```
pip install openai
```

Code 17: Installation of OpenAI

```
openai.api_key = GPT_API_KEY_VALUE
```

Code 18: API key setting up • Define a

- Define class named 'SASCodeAssistant' for OpenAI API.

```
import openai

class SASCodeAssistant:

    #initialize the class instance and accepts the OpenAI
    API key as a parameter.

     def __init__(self, api_key):

        self.api_key = api_key self.model_name = "gpt-3.5-
      turbo"
```

```
#set the model name and take parameter 'model_name' as
    customized model name.
def set_model_name(self, model_name):

     self.model_name = model_name

#generate_text method is used to generate text for users'
general questions.
def generate_text(self, prompt, max_tokens=100,
temperature=0.8):
     openai.api_key = self.api_key response =
     openai.Completion.create( engine=self.model
     _name, prompt=prompt,
     max_tokens=max_tokens,
     temperature=temperature
     )
     generated_text = response.choices[0].text.strip
         () return
     generated_text

#takes a SAS code as input and uses OpenAI's GPT model to
   check the syntax.
def check_syntax(self, code): openai.api_key =
     self.api_key response =
     openai.Completion.create( engine=self.mode
     l_name, prompt=code, max_tokens=0,
     temperature=0, n=1, stop=None,
     log_level="info"
     )
     messages = response.choices[0].message['
         logprobs']['token_logprobs']
     syntax_errors = [msg['token'] for msg in
            messages if msg['token'] == "ERROR"]
     return syntax_errors



#takes a prompt as input to generate SAS code based on the
given prompt.
def generate_code(self, prompt):
    openai.api_key = self.api_key response =
openai.Completion.create( engine=self.model_name,
prompt=prompt, max_tokens=100
     )
     generated_code = response.choices[0].text.strip()
return generated_code ...
```

Code 19: SASCodeAssistant Construction

- Initiate the SASCodeAssistant object as defined

```
sas_assistant = SASCodeAssistant()
```

Code 20: Initializing SASCodeAssistant object

- Apply GPT assistant by selecting code on the client-side and sending it to the server-side by pressing a hotkey 'ctrl + G'.

```
// Listen for hotkey events
document.addEventListener('keydown', (event) =>
  { // Check if the specified hotkey is pressed
  if (event.ctrlKey && event.key === 'G') {
    // Get the selected code
    const selectcode =
    window.getSelection().toString()
       ; const range = selection.getRangeAt(0); //
    Get the parent element of the selection range
    const parentElement =
       range.commonAncestorContainer ;
    // Find the closest <code> tag containing the
       selection
    const codeElement =
    parentElement.closest('sascode'
       );
  if(codeElement){
    // Get the content of the next sibling within the
       < code> tag
    const prompt
    =codeElement.nextSibling.textContent.trim(); }
    else{ const prompt = "";
    }
    data = {
        "sascode": selectcode,
        "user": $("#PersonID").get(0).title;,
        "timing": time(),
        "prompt": prompt
    }
    //  Send  the  code  to  the  server-side
  socket.emit('process_sas_request', data); }
});
...
```

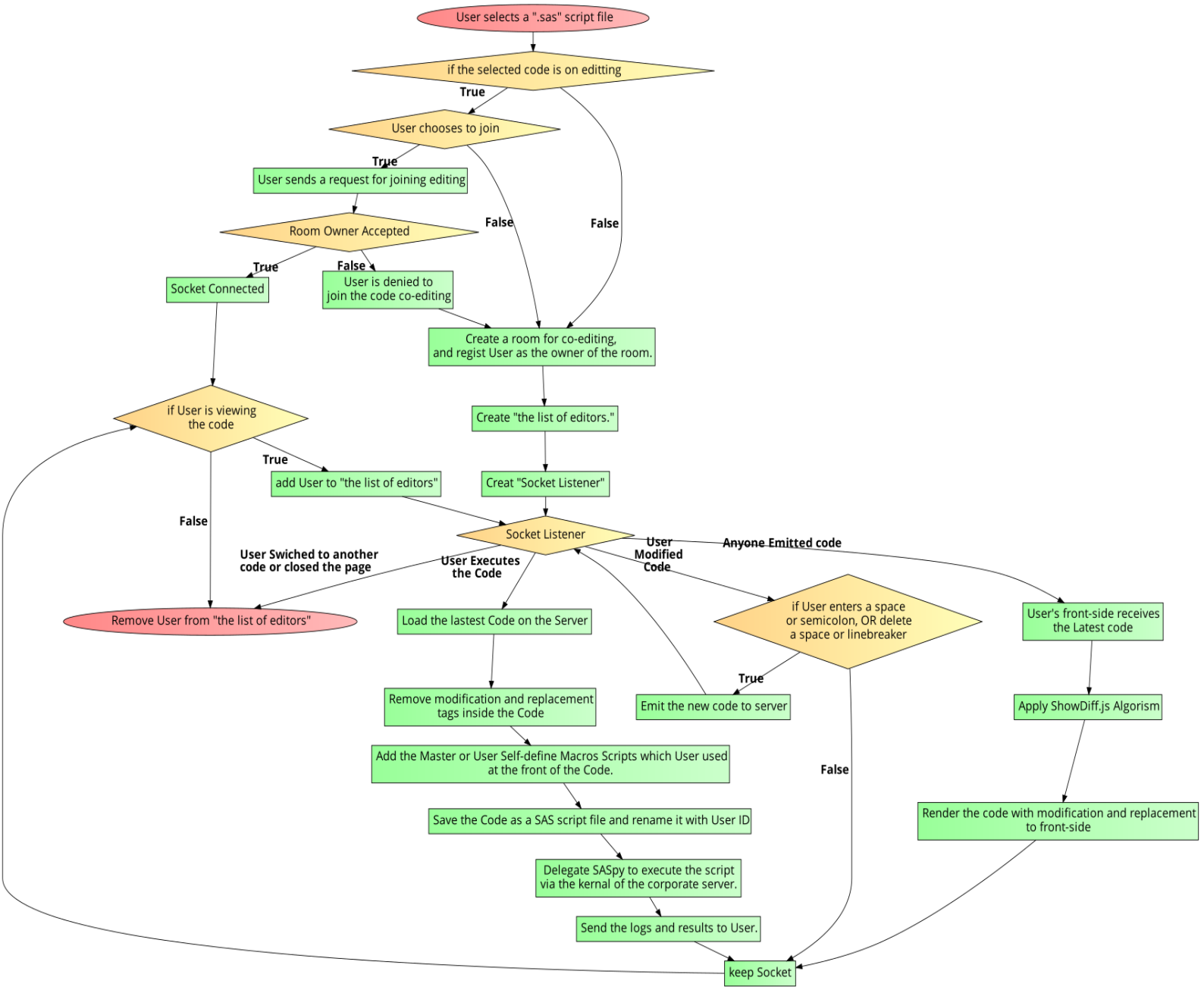
Code 21: Client-Side trigger define

• Apply sas_assistant object when receive the request from Client side.

```
...
@socketio.on('process_sas_request')
def process_sas_request(data):
    request_type = data.get('request_type')
    sascode = data.get('sascode')
    prompt = data.get('prompt')
    # Logic for syntax checking
    if request_type == 'check_syntax':
    #asking for code gramma check

       syntax_errors=sas_assistant.check_syntax( sa
       scode)
       response = {'syntax_errors': syntax_errors}
       # Logic for sascode checking

    elif request_type == 'generate_code': # Logic
    for sascode generation
        generated_code=sas_assistant.generate_code( prompt)

        response = {'generated_code': generated_code}
    else:
        generate_text=sas_assistant.generate_code(
        prompt)
        response = {'generate_text': generate_text}
    emit('sas_response', response)
    ...
```

Code 22: Server-side Response

OVERALL WORKFLOW



ADVANTAGES

The socket-based collaborative SAS editor brings several advantages that enable team members to work together effectively and enhance productivity. Here we also list out some advantages this editor will provide:

1.Real-time collaboration [Olson & Olson (2000)]: The editor's real-time synchronization feature ensures that team members can instantly see the edits made by others, facilitating real-time collaboration and discussions.

2.Conflict reduction [Damian & Zowghi (2003)]: With everyone able to see the sections being edited by others, the risk of conflicts and code overlaps is significantly reduced.

3.Version control integration [Bird et al. (2006)]: The editor can integrate with version control systems such as Git, allowing team members to track changes, roll back to previous versions, and view code history.

4.Instant communication [Gutwin & Greenberg (2002)]: The editor typically provides built-in chat or commenting functionality, enabling team members to communicate and discuss directly within the editing environment.

5.Cross-geographical collaboration [Li et al. (2013)]: Team members can work simultaneously in different locations around the globe, enabling global talent utilization and providing comprehensive and diverse solutions

CONCLUSION

In conclusion, the Socket-Based Collaborative SAS Editor provides a powerful solution for collaborative SAS programming. By leveraging socket-based communication, multiple users from different locations can seamlessly collaborate on SAS code development in real-time. The integration of the GPT API further elevates the capabilities of the editor by enabling intelligent assistance and code suggestions. This combination of collaborative features and AI-powered enhancements empowers SAS programmers to work more efficiently, solve problems effectively, and foster better communication and mentoring within the development team. It revolutionizes the way SAS code is developed and promotes efficient collaboration among programmers. It is an remarkable tool that empowers SAS developers to work together, leverage AI capabilities, and ultimately deliver superior SAS code solutions.

References

Bird, C., A. Gourley, P. Devanbu, A. Swaminathan & G. Hsu. 2006. Mining email social networks. In *Proceedings of the 2006 acm sigkdd international conference on knowledge discovery and data mining*, 439–448.

Brockman, Greg, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang & Wojciech Zaremba. 2016. Openai gym. *arXiv preprint arXiv:1606.01540* .

Damian, D. & D. Zowghi. 2003. The impact of communication dependencies on the maintenance of information systems. *Information and Software Technology* 45(14). 945–956.

Gutwin, C. & S. Greenberg. 2002. The effects of workspace awareness support on the usability of real-time distributed groupware. *ACM Transactions on Computer-Human Interaction* 9(2). 211–248.

Hickson, Ian. 2011. Websocket protocol. *RFC* 6455. https://tools.ietf.org/html/rfc6455.

Li, Q., S. Liu & X. Ye. 2013. Cross-geographical collaboration in open-source software development: An empirical study. *Information and Software Technology* 55(9). 1691–1705.

Lombardi, A. 2015. *Websocket: lightweight client-server communications*. O'Reilly Media, Inc.

Muller, G. L. 2014. Html5 websocket protocol and its application to distributed computing. *arXiv preprint arXiv:1409.3367* .

Olson, G. M. & J. S. Olson. 2000. Distance matters. *Human-Computer Interaction* 15(2-3). 139–178.

Perez, D., J. Xu & B. Livshits. 2020. Revisiting transactional statistics of highscalability blockchains. In *Proceedings of the acm internet measurement conference*, 535– 550.

Pimentel, V. & B. G. Nickerson. 2012. Communicating and displaying real-time data with websocket. *IEEE Internet Computing* 16(4). 45–53.

CONTACT INFORMATION

Contact the author at: Furan Wang.

Pfizer Research China

Pfizer China R&D Center Building No.3, Lane 60, Naxian Road, Shanghai, P.R.China 201210.

Email: Furan.wang@pfizer.com