Summary & Introduction

StudyLink is an app designed to help students easily make study buddies with peers from their university classes. By offering a way to find fellow classmates and reach out, StudyLink reduces the anxiety of asking for help in large lectures and creates a more personal and approachable environment for students who want to practice, collaborate, or seek help in their courses. Using the Canvas API, StudyLink automates the process of pulling student enrollment data and course information, making it effortless for students to discover peers in their classes for collaboration and study sessions.

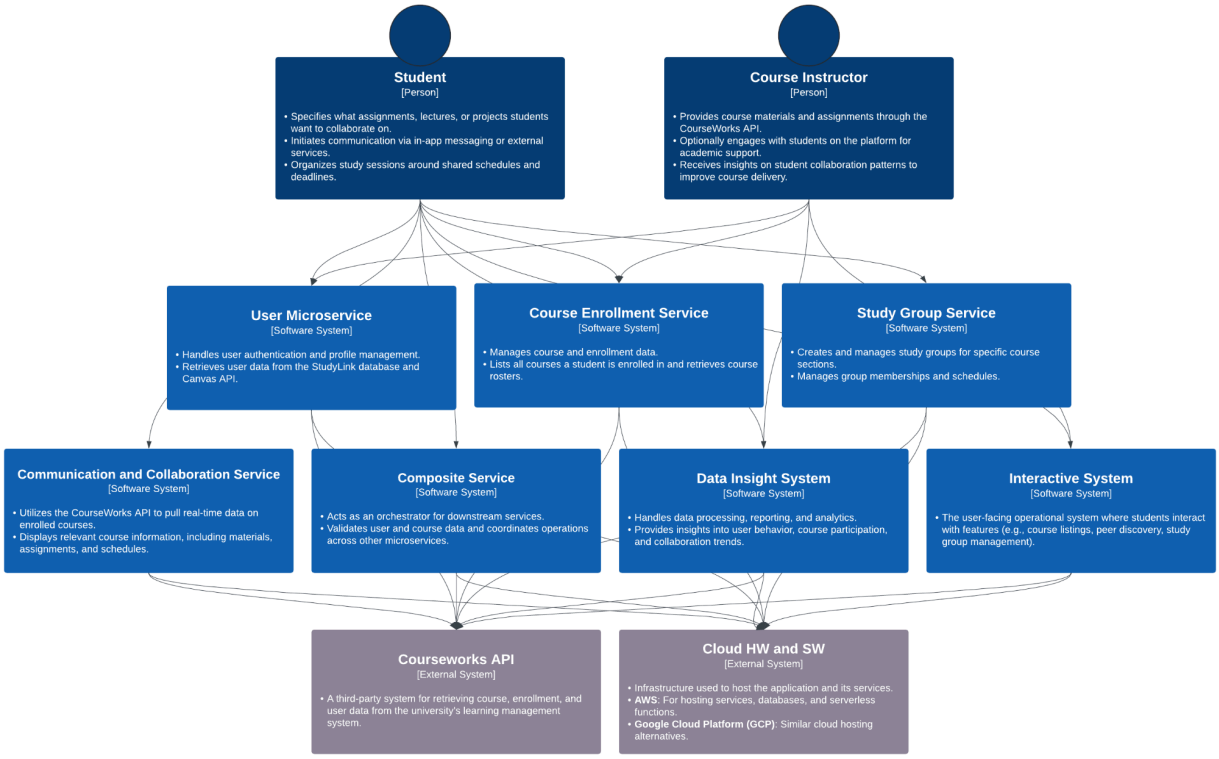
Some examples of the tasks StudyLink addresses are:

- Providing all of the courses in which a student is actively enrolled
- Students can see a list of classmates who are also using the app.
- Students can manage study groups by creating, joining, deleting, or editing details
- Facilitating communication between students via in-app messaging—between individual students or study groups
- A calendar that integrates key deadlines and events to help students manage their studying sessions.

StudyLink is a cloud-based app that delivers a significant improvement in how students connect for academic collaboration, making it easier to form study groups and engage with peers. The value to students is increased access to study support and reduced anxiety in large, impersonal lecture settings. The benefit to universities is that StudyLink fosters a more supported, connected, and collaborative student body.

Solution Overview

Domains



Overall Solution

StudyLink is a cloud-based platform designed to foster academic collaboration by connecting students within their university classes. The app addresses the challenges students face in large, impersonal lecture settings by providing a seamless way to discover peers for study groups, collaborate on assignments, and exchange ideas. StudyLink integrates with the university's learning management system (Canvas API) to automate data retrieval, ensuring minimal manual effort for users. Through its modular design, StudyLink offers robust support for user management, course enrollment, study group formation, communication, and scheduling.

Major Systems and Subsystems

<u>User Service</u>

Students log in using their university credentials and the service retrieves their personal information (e.g., name, email) from the Canvas API. Students can update their profile settings directly in StudyLink.

- Handles user authentication and manages user profiles.
- Retrieves user data from Canvas API and StudyLink's database.
- Updates user-specific settings and preferences.

Course Enrollment Service

Students can view a list of all their enrolled courses, fetched from the Canvas API. Within a selected course, the service provides a filtered list of classmates using StudyLink.

- Automates the retrieval of course and enrollment data.
- Fetches course lists for users from Canvas.
- Lists classmates (and their IDs)enrolled in specific courses.

Study Group Service

Students can create a study group for a course if none exists or join an existing group.

- Facilitates the creation and management of study groups.
- Create study groups for courses if none exist.
- Manages group memberships and meeting details.

Communication and Collaboration Service

Students can send messages to classmates, either one-on-one or in study groups, to discuss specific assignments, share resources, and organize group study sessions.

- Enables seamless interaction among users.
- Provide in-app messaging and group chat functionality.
- Allows notifications through email or external messaging platforms.

Composite Service

When students create a study group, join a discussion, or send a message, the Composite Service coordinates these operations by interacting with relevant downstream services (e.g., verifying enrollment and updating group memberships).

- Acts as an orchestrator for downstream microservices.
- Validates user and course data for study group creation.
- Handles asynchronous requests to improve performance.

Data Insight System

Students can view their participation metrics, such as study groups created or messages exchanged.

- Provides analytics and insights into user behavior and engagement.
- Analyzes student participation in study groups.
- Identifies collaboration trends to improve the app experience.

Interactive System

Students interact with all app features (e.g., viewing courses, joining groups, messaging) through a user-friendly interface. They can sync deadlines and organize study sessions directly through the interface.

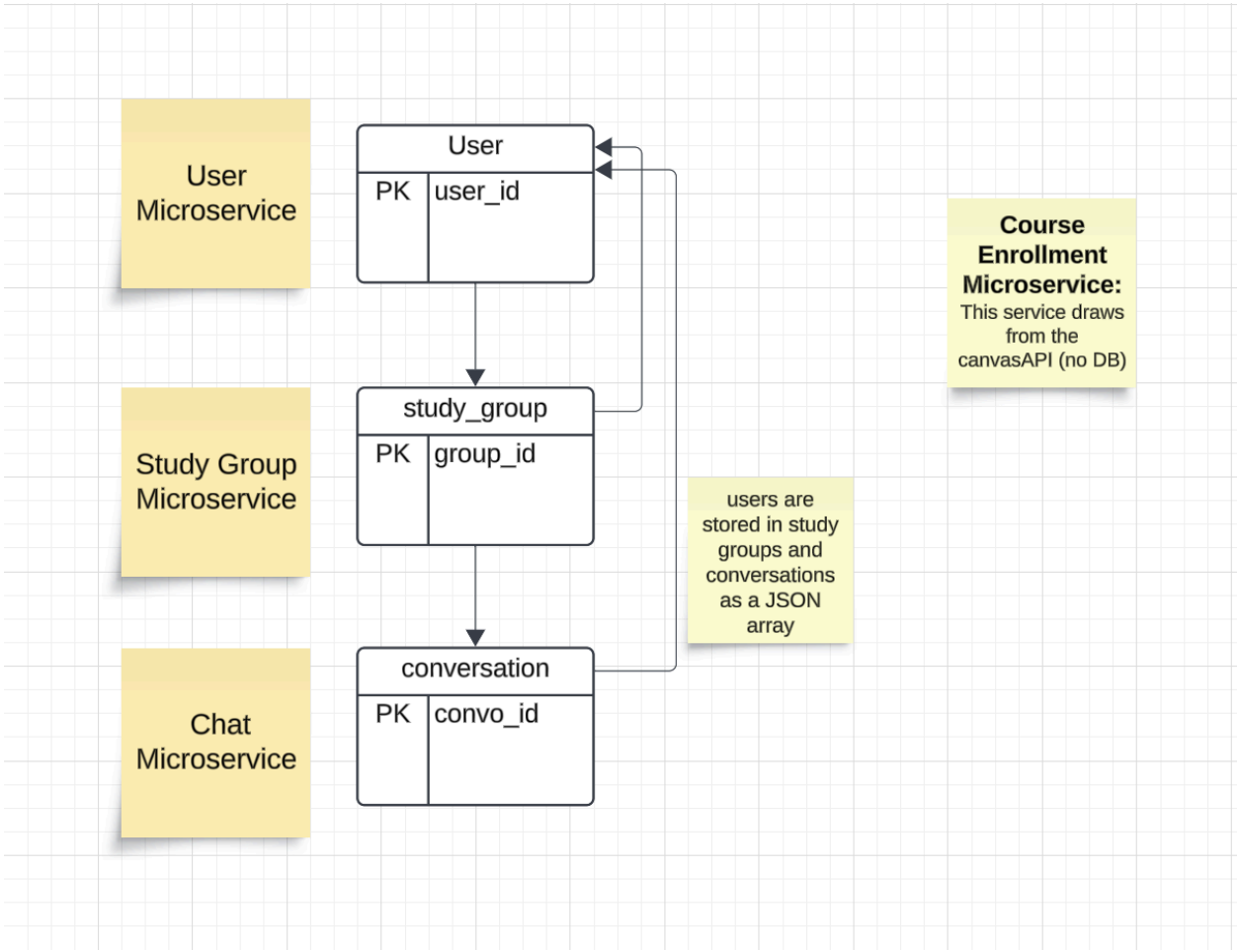
- Front-end system that provides the primary user interface.
- Enables students to view courses, find classmates, and manage study groups.
- Displays schedules, deadlines, and messaging functionality.

External Systems

- Canvas API: Integrates with the university's learning management system to fetch course and enrollment data.
- Cloud Infrastructure: Hosts StudyLink's services, ensuring scalability and availability.

Resource Model

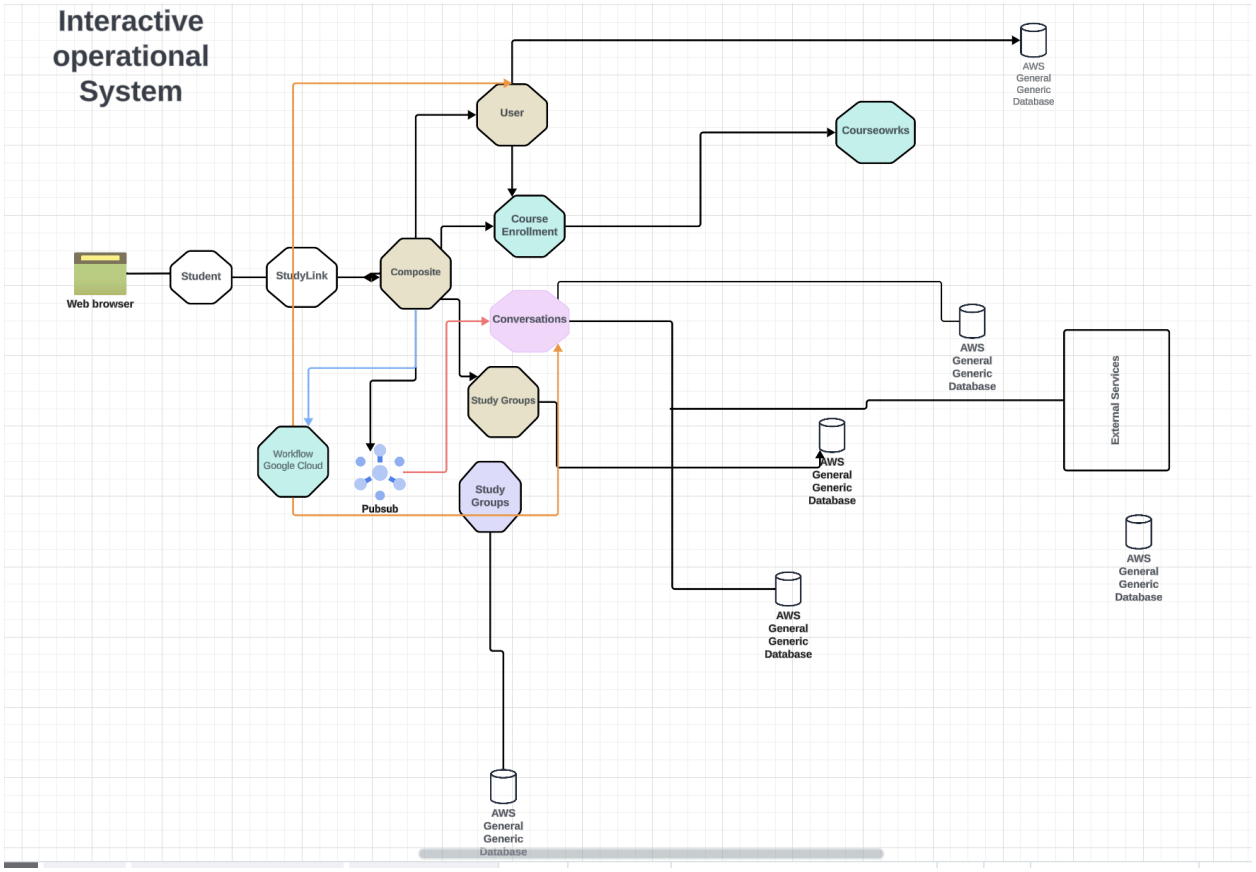
[Resource Model Link]



Description

- User: CU student in courseworks that has signed up
- Study group: study groups for students in the same course
- Chat: private and group conversations

Interactive and Operations System

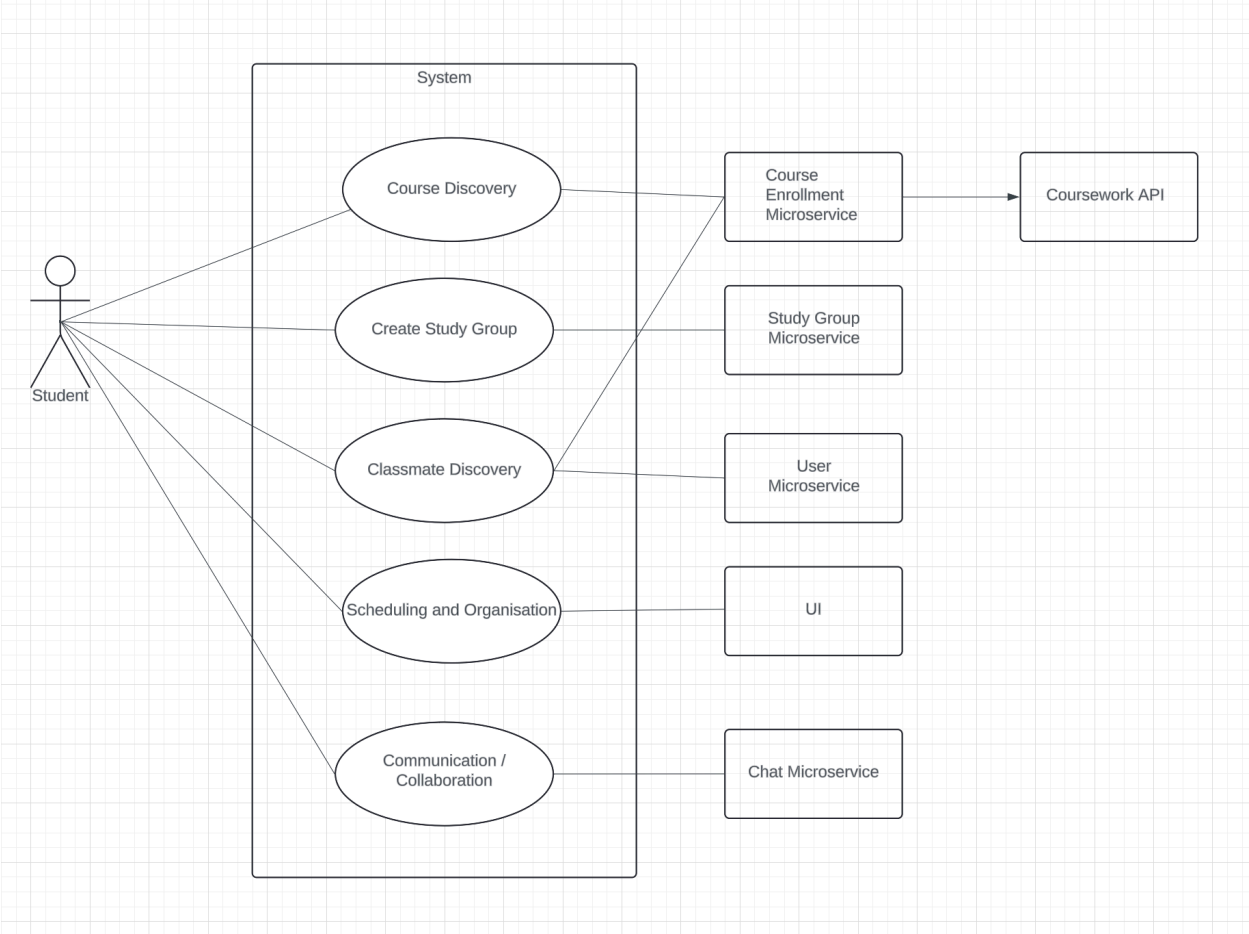


System Overview:
Begin by logging in through the browser user interface to access our application, which offers a variety of functionalities:

1) Course Enrollment: Retrieve course details and enrollment data from the Courseworks API. Store the info in the application database for easy access and management.
2) User: Access and manage the student's personal information, including name, ID, and contact details. Obtain information from the Canvas API and the application database.
3) Conversations/Messaging Service: Allow students to create conversations within a group chat or direct message format.
4) Student Groups: Help students of the same course to create study groups and study together

Use Case Diagram

[Diagram Link]



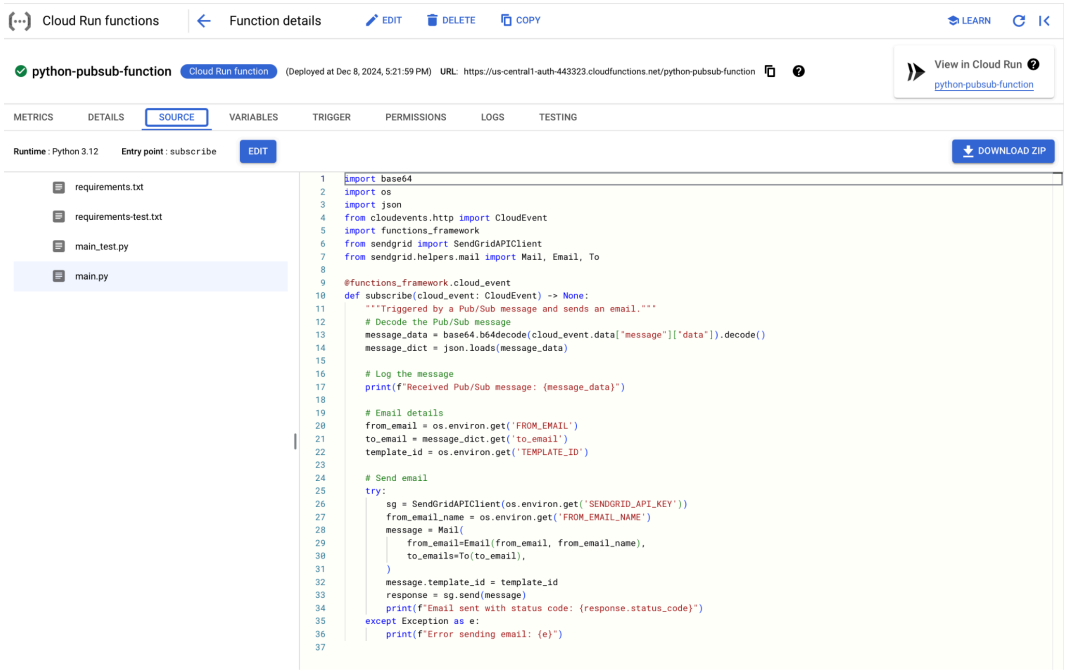
Microservices

User Profile Microservice

The user profile microservice is responsible for pulling user-specific information from Canvas API and the application database.

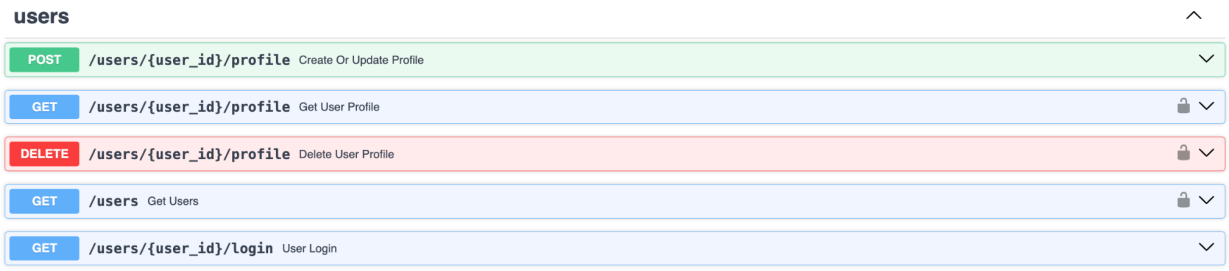
When creating an account for StudyLink, the user authenticates their account via Google Login and a Canvas token. User information—full name, email address, enrolled courses, and pronouns—are retrieved from the user's Canvas profile. As user profile information can be updated in Canvas, and active courses change every semester, the service updates this data accordingly. The User microservice contains an authentication section in the middleware that checks the Google Token and the JWT Token passed in the headers of PUT, POST and DELETE requests, as well as some GET requests. This section will be used by the Composite microservice to verify authentication for PUT, POST and DELETE requests for all microservices. It is also an additional security check, on top of what is implemented in the API Gateway.

Upon signing up to the application, the user will receive an automatic email notification confirming they have signed up. This FaaS was implemented through Google Cloud Run Functions. The email is sent via an integration with SendGrid, which sends the user a dynamic email template, which updates based on the name of the new user:



The User Profile Microservice can create and update a user profile, retrieve a single user profile or query for a list of students by name or course, and delete a profile:

OpenAPI Documentation



Create or Update Profile (POST)

`POST /users/{user_id}/profile`

```
curl -X POST "http://localhost:8000/users/{user_id}/profile" \
-H "Authorization: Bearer <JWT_TOKEN>" \
-H "Content-Type: application/json" \
-d '{ "token": "<PROFILE_TOKEN>" }'
```

Replace `{user_id}` with the user ID, `<JWT_TOKEN>` with the actual token, and `<PROFILE_TOKEN>` with the profile token.

Get User Profile (GET)

`GET /users/{user_id}/profile`

```
curl -X GET "http://localhost:8000/users/{user_id}/profile" \
-H "Authorization: Bearer <JWT_TOKEN>"
```

Replace `{user_id}` and `<JWT_TOKEN>` with the user ID and token.

Delete User Profile (DELETE)

`DELETE /users/{user_id}/profile`

```
curl -X DELETE "http://localhost:8000/users/{user_id}/profile" \
-H "Authorization: Bearer <JWT_TOKEN>"
```

Replace `{user_id}` and `<JWT_TOKEN>` with the user ID and token.

Get Users List (GET)

`GET /users`

```
curl -X GET
"http://localhost:8000/users?skip=0&limit=10&name=<NAME>&course=<COURSE>" \
-H "Authorization: Bearer <JWT_TOKEN>"
```

Replace `<NAME>` and `<COURSE>` with search parameters if needed. You can omit them for general listing. Replace `<JWT_TOKEN>` as required.

User Login (GET)

`GET /users/{user_id}/login`

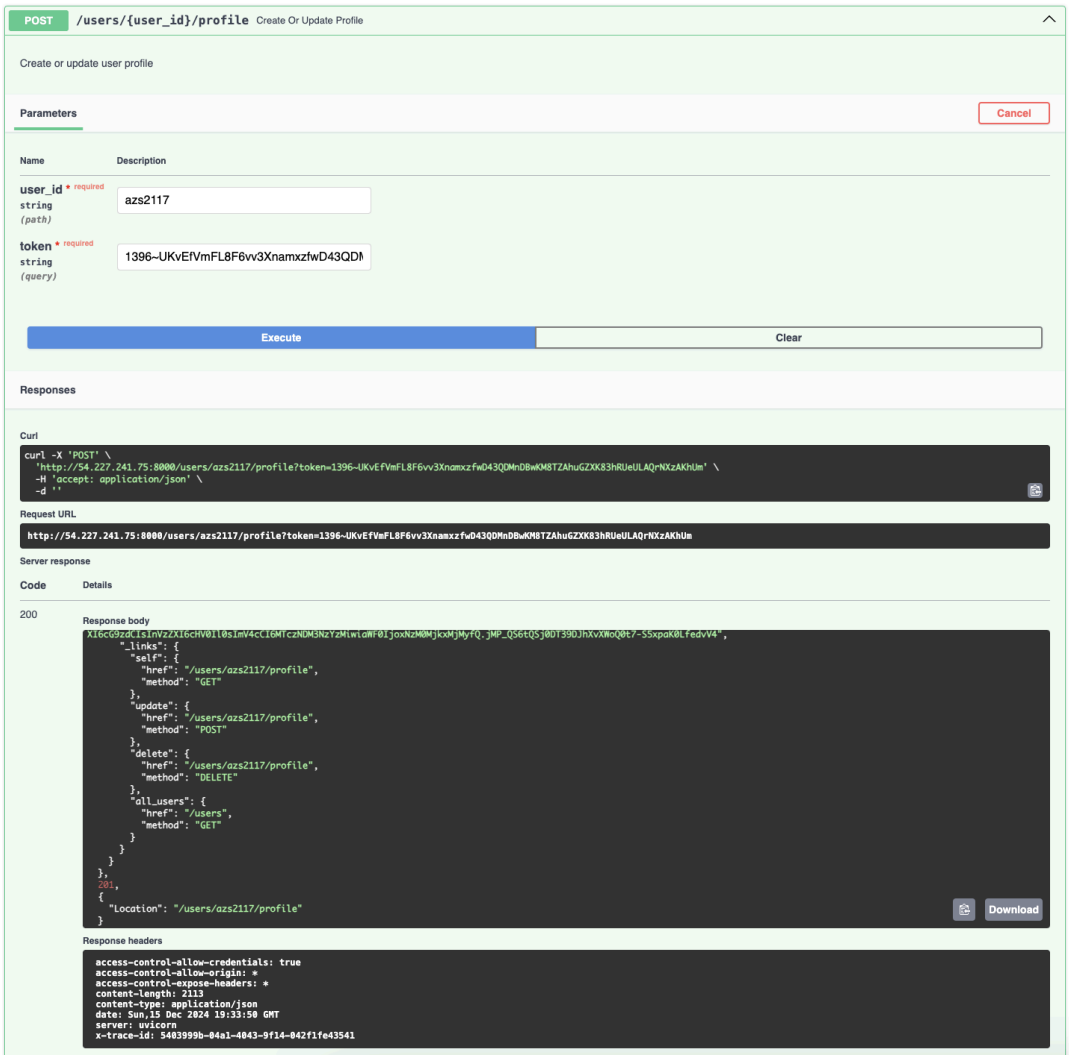
```
curl -X GET "http://localhost:8000/users/{user_id}/login" \
-H "Authorization: Bearer <JWT_TOKEN>"
```

Replace `{user_id}` and `<JWT_TOKEN>` with appropriate values.

FastAPI Responses

- Uses public IP address from EC2 instance
- Demonstrates connection to Canvas and StudyLink database
- HATEOAS and links
- Correlation ID and propagation (`x-trace-id`)

Example of creating a user profile (POST):

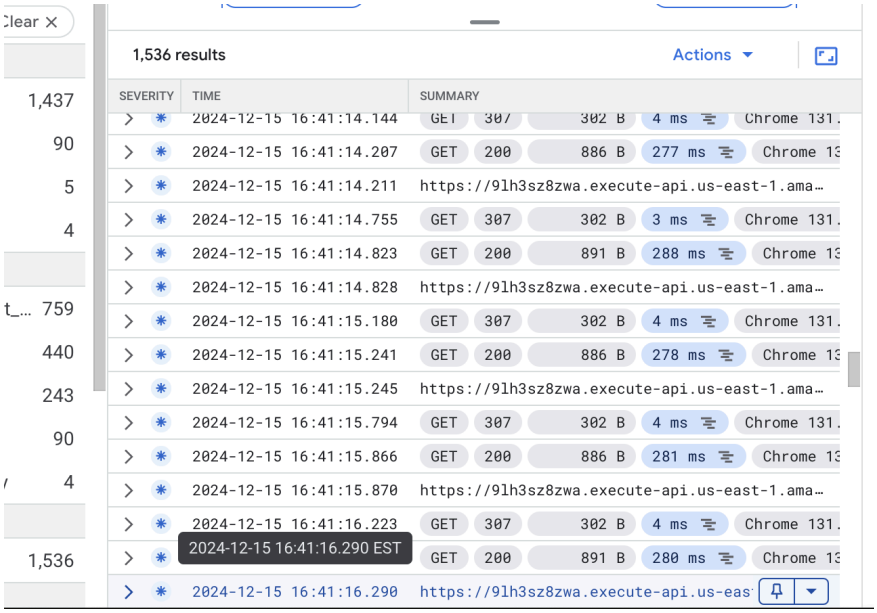


Logging and Tracing

```python
@app.middleware("http")
async def log_requests(request: Request, call_next):
    trace_id = str(uuid.uuid4())  # Generate a unique trace ID for the request
    request.state.trace_id = trace_id
    logging.info(f"TRACE_ID={trace_id} - Incoming Request: {request.method} {request.url}")

    start_time = time.time()
    response = await call_next(request)
    processing_time = time.time() - start_time

    logging.info(
        f"TRACE_ID={trace_id} - Completed Request: {request.method} {request.url} "
        f"with Status {response.status_code} in {processing_time:.4f}s"
    )
    response.headers["X-Trace-Id"] = trace_id
    return response
```

Example of EC2 Instance (AWS)

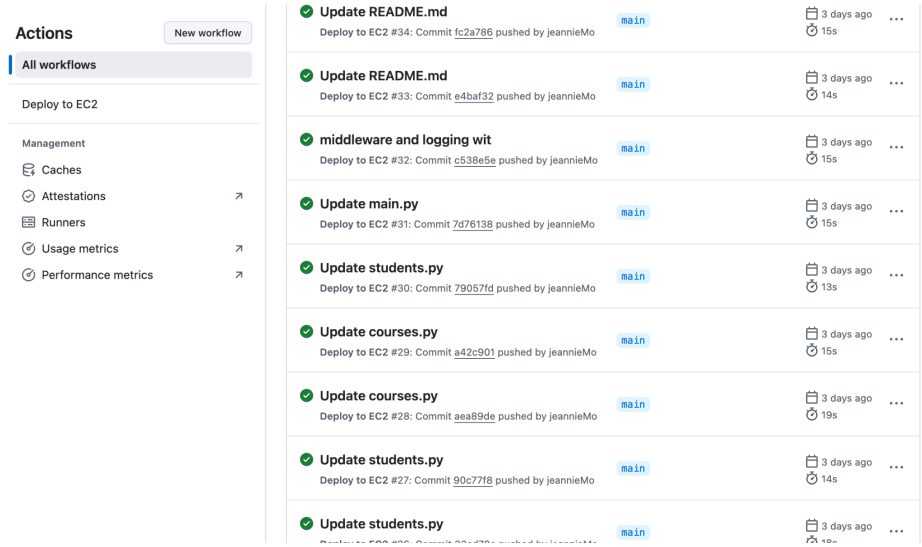
Example of GCP logging



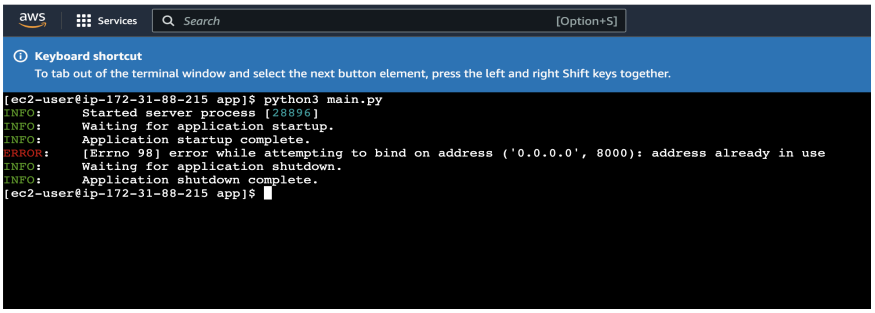
Course Enrollment Service

- **Functionality**: This microservice pulls the enrollment data from the university's system via the Canvas API. It is responsible for listing all the courses a student is enrolled in and all the students enrolled in a course. Accepts GET requests.
    - Retrieve student-specific enrollment data when given a uni and Canvas token.
    - Retrieve roster of all students in a course when given course code and Canvas token.
    - Update and manage course data for each student.
    - Provide an interface for other services to access a student's course list.
- **How It Works**: When a student logs into StudyLink, this service fetches all the courses in which the student is enrolled in. When a student clicks to create a study group, they will see a list of their classes and subsequently their peers who are also enrolled in StudyLink (using information from the User microservice database).

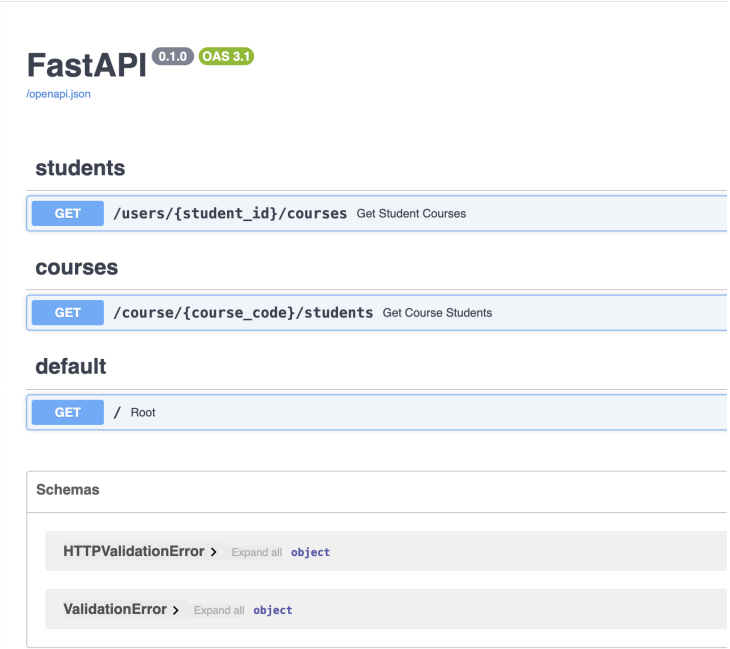
Incorporates github workflow that deploys a microservice on commit:



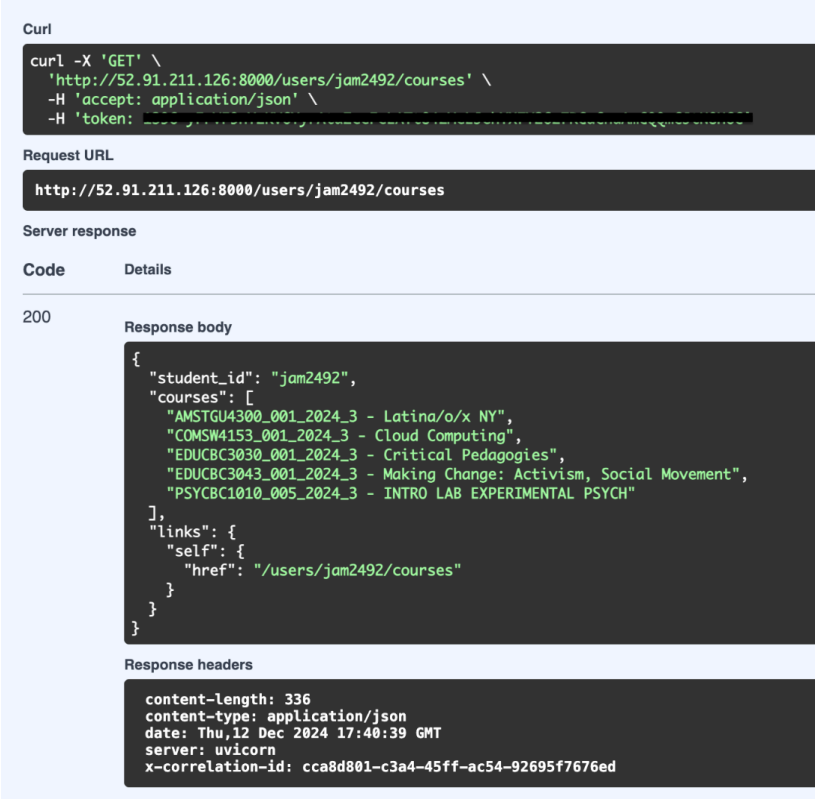
Showing application running in AWS:



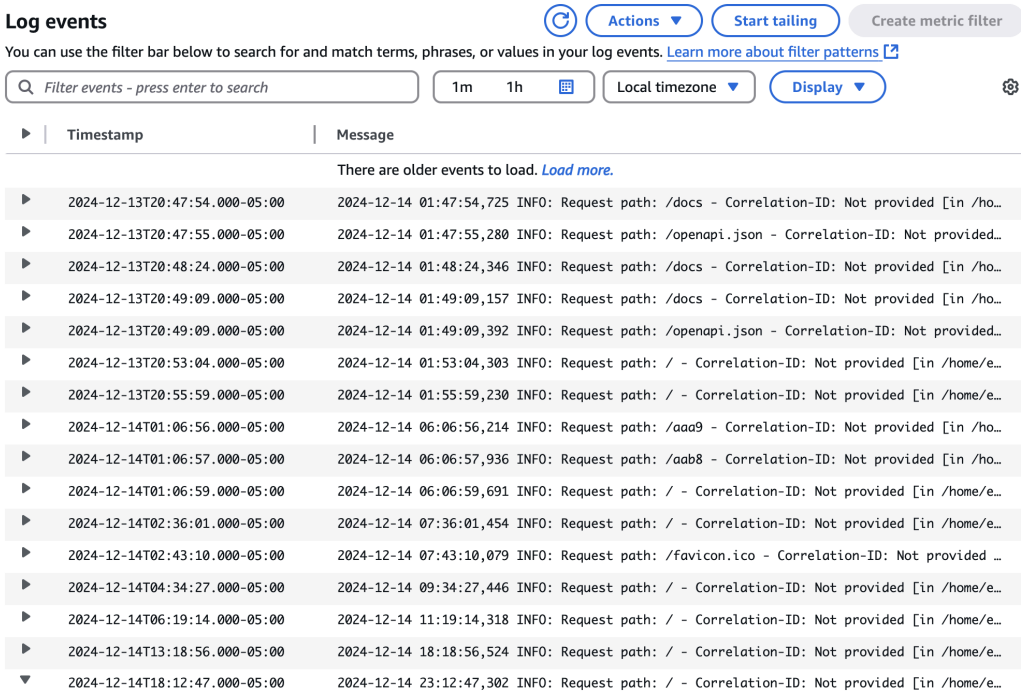
FastAPI entry page, using public IP address from EC2 instance + calls on FastAPI to show connection to Canvas and Database:



Showing HATEOAS Links and Correlation ID propagation and tracking in response:



Logging for Microservice through CloudWatch:

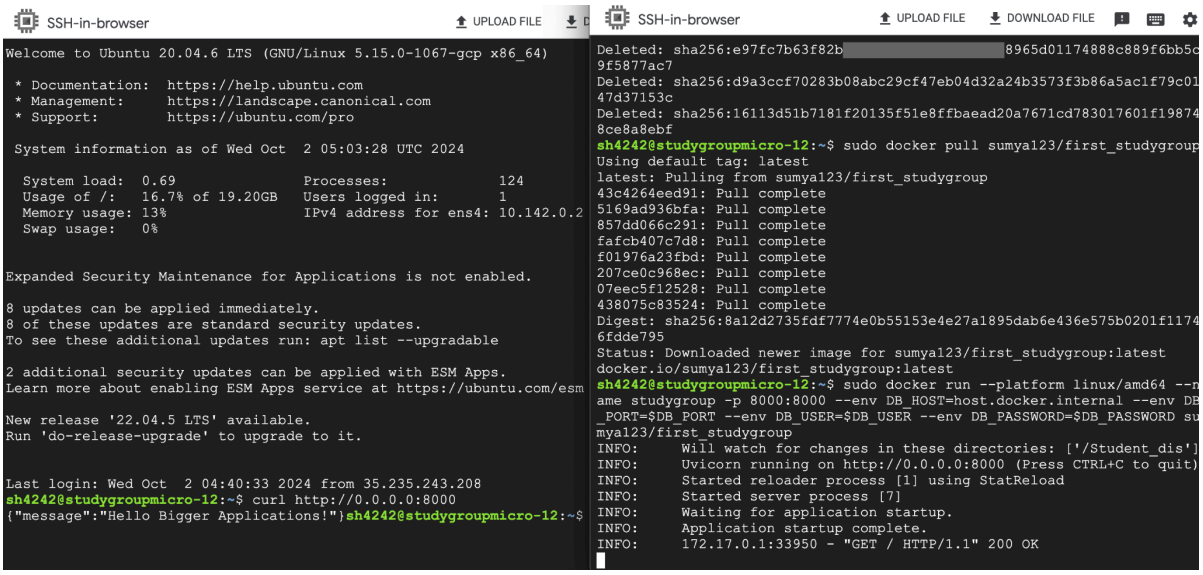


Study Group Service

- The Study Group Microservice is a central component of the StudyLink application, which aims to help students bring students together, allowing them to study together.
- Through this microservice, it is possible to:
    - Create new study groups
    - Update existing groups with relevant details
    - Delete groups no longer needed
    - Retrieve information about all study groups or a specific one

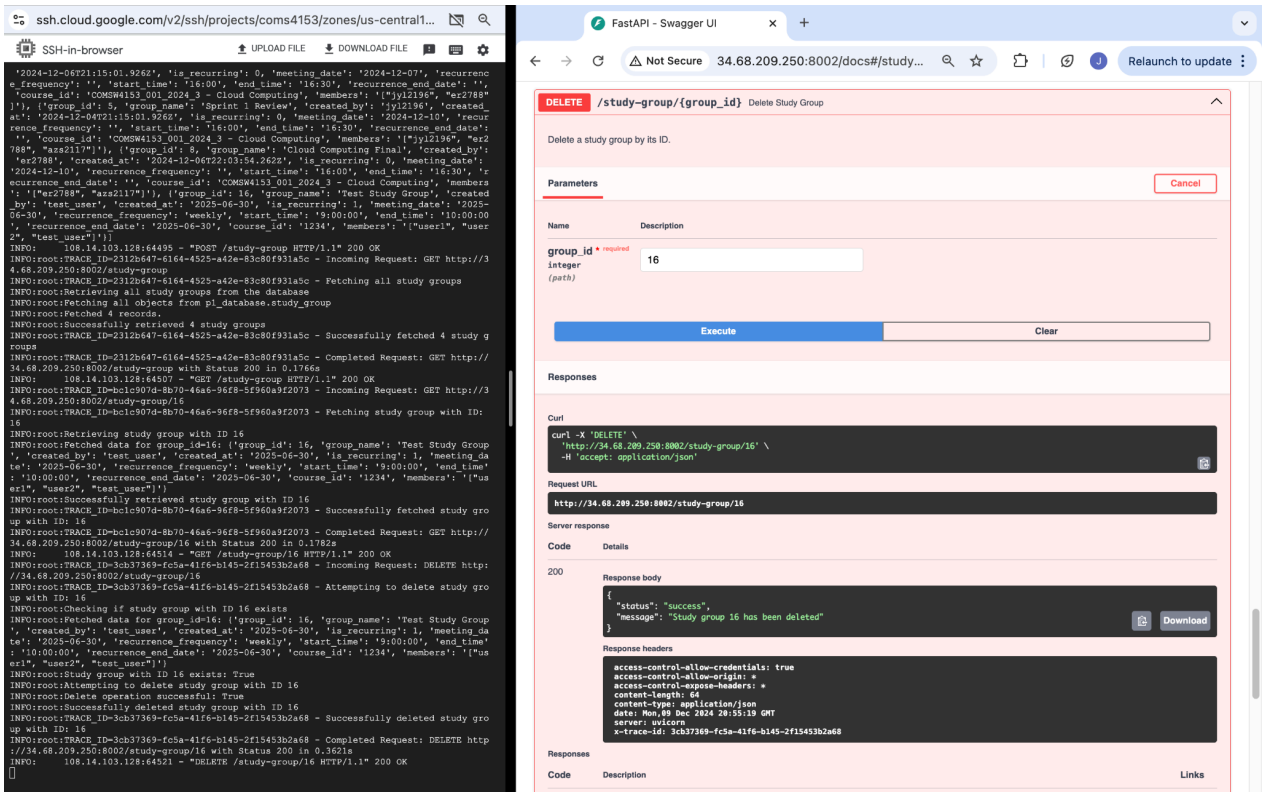
Used Docker to containerize the whole application and deploy on Google Cloud (Ubuntu Platform)

Docker Deployment



Log and trace, Correlation ID and propagation, in a cloud's logging tools

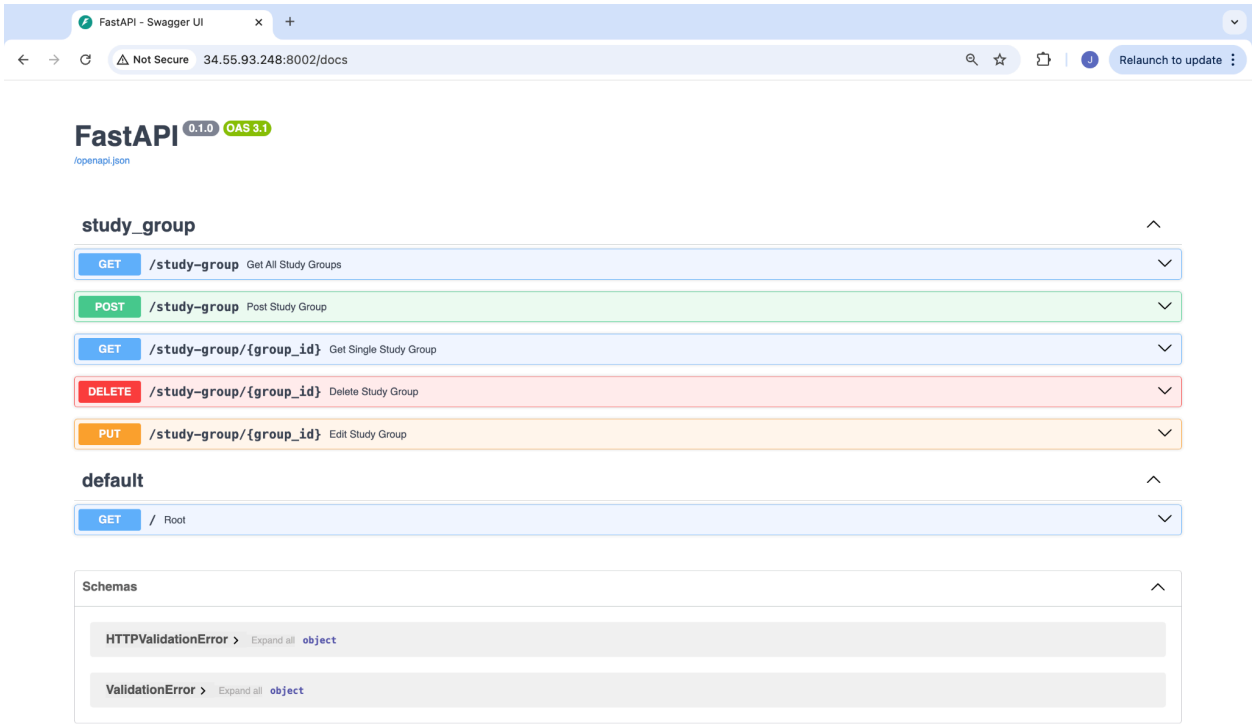
This shows the microservice running on GCP with logging and correlation ID propagation. The left shows the cloud logging, and the right shows the OpenAPI docs and most recent request (DELETE).



Logs are also in this format:

```
logging.info(f"TRACE_ID={trace_id} - Incoming Request: {request.method} {request.url}")

start_time = time.time()
response = await call_next(request)
processing_time = time.time() - start_time

logging.info(
    f"TRACE_ID={trace_id} - Completed Request: {request.method} {request.url} "
    f"with Status {response.status_code} in {processing_time:.4f}s"
)
response.headers["X-Trace-Id"] = trace_id  # Include trace ID in response headers
```

OpenAPI documents



HATEOAS and support for links

This demonstrates a few success calls for post, put and get

POST


```
{
  "status": "success",
  "message": "Group 'Test Study Group' created successfully.",
  "group_id": 17,
  "data": {
    "group_name": "Test Study Group",
    "created_by": "test_user",
    "created_at": "2025-06-30",
    "course_id": "1234",
    "is_recurring": true,
    "meeting_date": "2025-06-30",
    "recurrence_frequency": "weekly",
    "recurrence_end_date": "2025-06-30",
    "start_time": "9:00:00",
    "end_time": "10:00:00",
    "members": [
      "user1",
      "user2",
      "test_user"
    ]
  },
  "_links": {
    "self": {
      "href": "/study-group/17"
    },
    "list": {
```


PUT


```
{
  "status": "success",
  "message": "Study group 18 has been updated successfully",
  "updated_fields": {
    "group_name": "Updated Study Group Name",
    "is_recurring": false,
    "meeting_date": "2025-06-30",
    "recurrence_end_date": "2025-07-30",
    "start_time": "09:30:00",
    "end_time": "10:30:00",
    "members": "[\"user3\", \"user4\"]"
  },
  "_links": {
    "self": {
      "href": "/study-group/18"
    },
    "list": {
      "href": "/study-group"
    },
    "edit": {
      "href": "/study-group/18",
      "method": "PUT"
    },
    "delete": {
      "href": "/study-group/18",
      "method": "DELETE"
```

<table>
<tr>
<td>

<pre>
    "href": "/study-group"
   }
  }
}
</pre>

</td>
<td>

<pre>
   }
  }
}
</pre>

</td>
</tr>
<tr>
<td>

GET

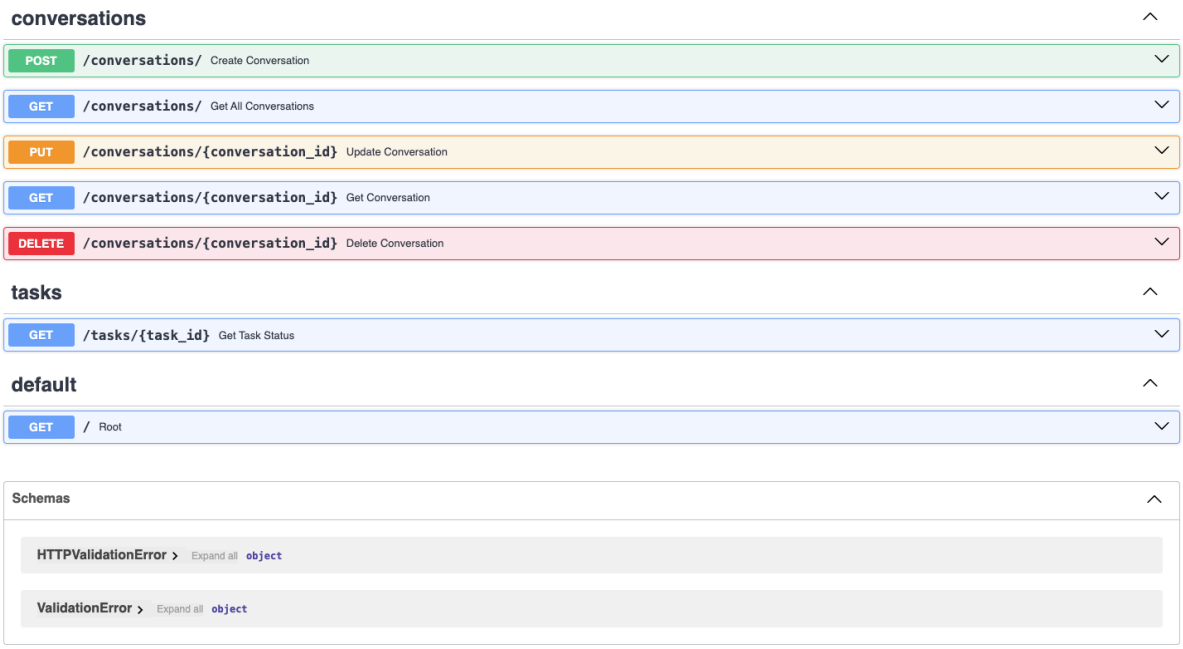
<pre>
{
  "status": "success",
  "data": [
    {
      "group_id": 4,
      "group_name": "Cloud Computing",
      "created_by": "jyl2196",
      "created_at": "2024-12-06T21:15:01.926Z",
      "is_recurring": false,
      "meeting_date": "2024-12-07",
      "recurrence_frequency": "",
      "start_time": "16:00",
      "end_time": "16:30",
      "recurrence_end_date": "",
      "course_id": "COMSW4153_001_2024_3 - Cloud
Computing",
      "members": [
        "jyl2196",
        "er2788"
      ],
      "_links": {
        "self": {
          "href": "/study-group/4"
        },
        "edit": {
          "href": "/study-group/4",
          "method": "PUT"
        },
        "delete": {
          "href": "/study-group/4",
          "method": "DELETE"
        }
      }
    },
    …the rest of the groups
  ],
  "message": "Fetched 3 study group(s)"
}
</pre>

</td>
<td></td>
</tr>
</table>

Chat Service

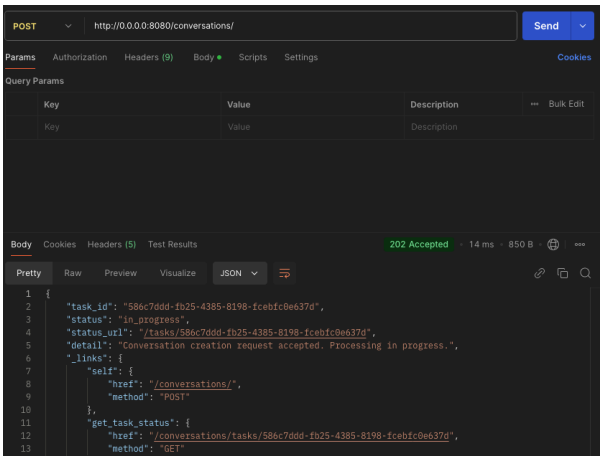
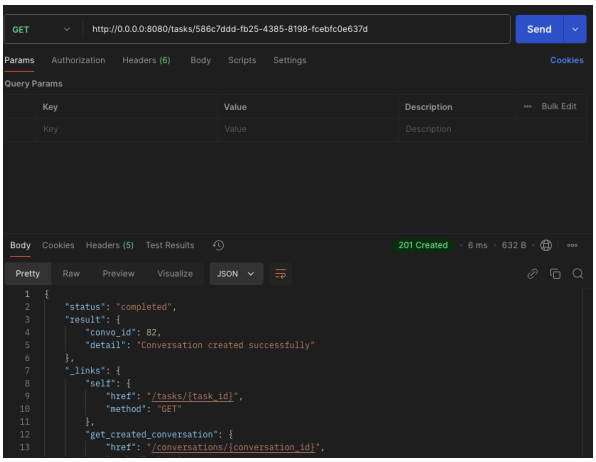
- **Functionality**: The chat microservice allows the creation, modification and retrieval of conversation details.
    - Create Conversations: Users can initiate new conversations, whether it's a one-on-one chat or a group discussion. The service generates unique identifiers for each conversation, ensuring easy access and management.
    - Retrieve Conversation Details: The microservice provides endpoints to fetch conversation details by convo ID / retrieve all convo details.
    - Modify Conversations: Users' conversation will be updated when new convos occur. This includes adding or removing participants, changing the conversation title, or simply having new messages .
    - Delete Conversation Details: The microservice provides endpoints to delete conversation details by convo ID.
- **How it works:** By supporting these functionalities, the chat microservice fosters dynamic interactions between users

OpenAPI documents



For POST, implement 201 created with a link header

When creating a conversation successfully, return 201 status code.

Status Code 202 - Create Request Accepted	Status Code 201 - Conversation Created
	

Asynchronous Execution Pattern

Create Conversation Response


```
{
  "task_id": "7107cc35-03c3-4fd0-be38-ebaba2f99dab",
  "status": "in_progress",
  "status_url": "/tasks/7107cc35-03c3-4fd0-be38-ebaba2f99dab",
  "detail": "Conversation creation request accepted. Processing in progress.",
  "_links": {
    "self": {
      "href": "/conversations/",
      "method": "POST"
    },
    "get_task_status": {
      "href": "/conversations/tasks/7107cc35-03c3-4fd0-be38-ebaba2f99dab",
      "method": "GET"
    },
    "get_created_conversation": {
      "href": "/conversations/{conversation_id}",
      "method": "GET"
    },
    "delete_conversation": {
      "href": "/conversations/{conversation_id}",
      "method": "DELETE"
    },
    "list_conversations": {
      "href": "/conversations/",
      "method": "GET"
    },
    "create_conversation": {
      "href": "/conversations/",
```


Get Create Conversation Task Status


```
{
  "status": "completed",
  "result": {
    "convo_id": 82,
    "detail": "Conversation created successfully"
  },
  "_links": {
    "self": {
      "href": "/tasks/7107cc35-03c3-4fd0-be38-ebaba2f99dab",
      "method": "GET"
    },
    "get_created_conversation": {
      "href": "/conversations/82",
      "method": "GET"
    },
    "delete_conversation": {
      "href": "/conversations/82",
      "method": "DELETE"
    },
    "list_conversations": {
      "href": "/conversations/",
      "method": "GET"
    },
    "create_conversation": {
      "href": "/conversations/",
      "method": "POST"
    }
  }
}
```

"method": "POST" } } } STATUS: 202 Accepted	STATUS: 201 Created

Pagination for GET /conversations

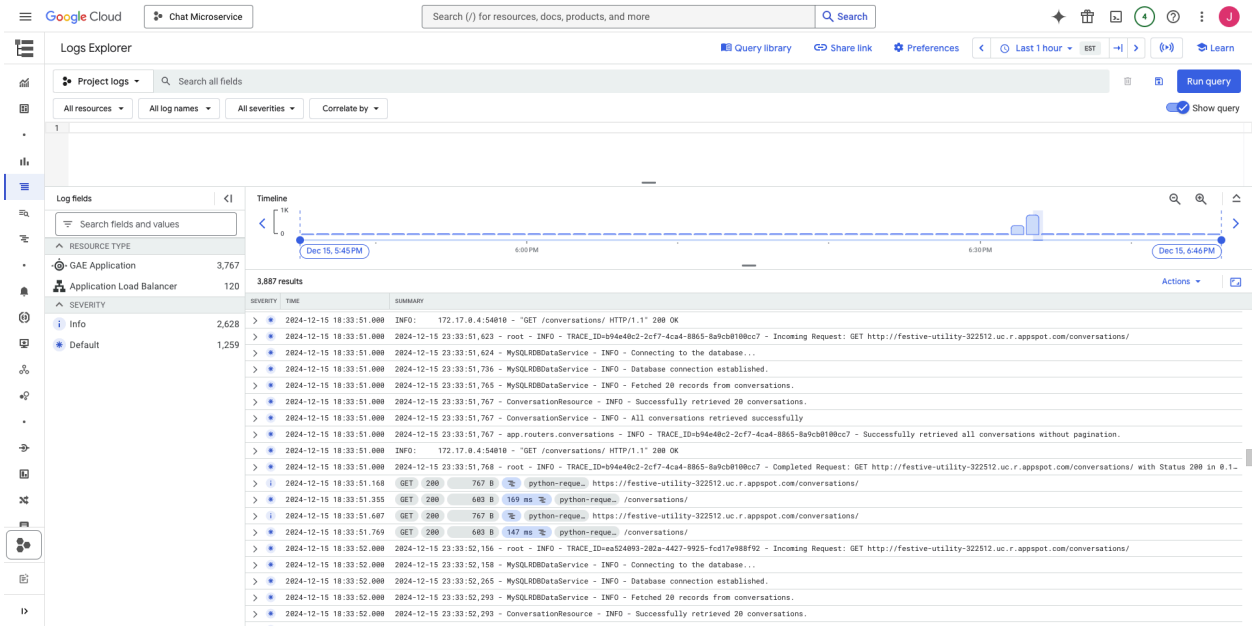
GET conversations/?page=1&limit=3	GET conversations/?page=2&limit=2

{
"detail": [
{
"convo_id": 63,
"name": "Jessica Liang - jyl2196 - Emanuela Romano - er2788",
"participants": "[\"er2788\", \"jyl2196\"]",
"messages": "[{\"text\": \"Hi Jess!\", \"sender\": \"er2788\", \"timestamp\": \"17:04:07\"}]",
"isGroup": 0
},
{
"convo_id": 64,
"name": "Cloud Computing",
"participants": "[\"jyl2196\", \"er2788\"]",
"messages": "[{\"text\": \"Hi group!\", \"sender\": \"er2788\", \"timestamp\": \"17:04:15\"}]",
"isGroup": 1
},
{
"convo_id": 65,
"name": "Sprint 1 Review",
"participants": "[\"jyl2196\", \"er2788\", \"azs2117\"]",
"messages": "[]",
"isGroup": 1
}
],
"total": 14,
"page": 1,
"limit": 3,
"total_pages": 5
}
	{
	"detail": [
	{
	"convo_id": 65,
	"name": "Sprint 1 Review",
	"participants": "[\"jyl2196\", \"er2788\", \"azs2117\"]",
	"messages": "[]",
	"isGroup": 1
	},
	{
	"convo_id": 66,
	"name": "Coding Workshop",
	"participants": "[\"John Doe\", \"Jane Smith\", \"Bob Johnson\", \"CurrentUser\"]",
	"messages": "[{\"text\": \"Welcome to our project group!\", \"sender\": \"CurrentUser\", \"timestamp\": \"9:00 AM\"}]",
	"isGroup": 1
	}
	],
	"total": 14,
	"page": 2,
	"limit": 2,
	"total_pages": 7
	}

Logging + Correlation ID and propagation

```
2024-12-14 08:59:18,446 - root - INFO - TRACE_ID=08f9744a-2d8d-40b3-bc43-4fd44d6088f0 - Incoming Request: GET http://0
.0.0.0:8080/conversations/72
2024-12-14 08:59:18,447 - MySQLRDBDataService - INFO - Connecting to the database...
2024-12-14 08:59:18,492 - MySQLRDBDataService - INFO - Database connection established.
2024-12-14 08:59:18,500 - MySQLRDBDataService - INFO - Successfully fetched a record from conversations.
2024-12-14 08:59:18,501 - ConversationResource - INFO - Conversation with ID 72 retrieved successfully.
2024-12-14 08:59:18,501 - ConversationService - INFO - Conversation with ID 72 retrieved successfully
2024-12-14 08:59:18,501 - app.routers.conversations - INFO - TRACE_ID=08f9744a-2d8d-40b3-bc43-4fd44d6088f0 - Successfu
lly retrieved conversation with ID 72
2024-12-14 08:59:18,501 - root - INFO - TRACE_ID=08f9744a-2d8d-40b3-bc43-4fd44d6088f0 - Completed Request: GET http://
0.0.0.0:8080/conversations/72 with Status 200 in 0.0555s
INFO:     127.0.0.1:49781 - "GET /conversations/72 HTTP/1.1" 200 OK
```

Demonstrate in a cloud's logging tools



Composite Service

API Lists and description

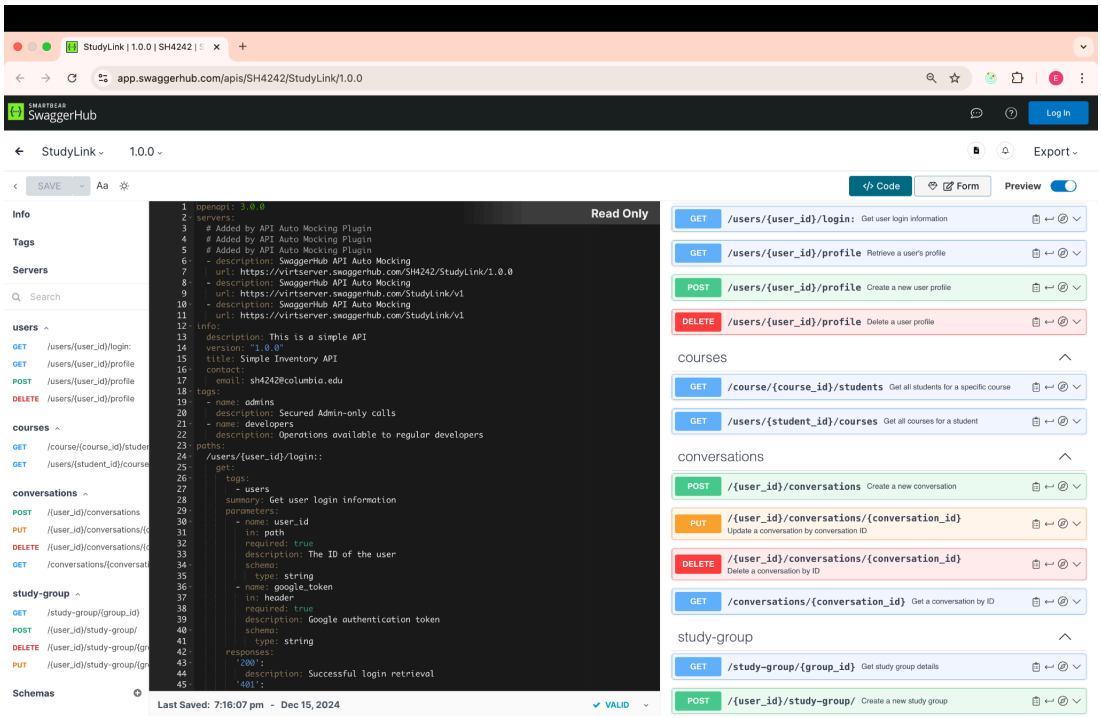
1. **User Service**
    a. Get /StudyLink/v1/users/{user_id}/login → call the user microservice logging endpoint
    b. Get /StudyLink/v1/users/{user_id}/profile → get request, has JWT and Google Token as a header, pass along to the user service to verify
    c. Post /StudyLink/v1/users/{user_id}/profile → post request, has query parameters (canvas token) and Google Token
    d. Delete /StudyLink/v1/users/{user_id}/profile → delete request, has JWT and Google Token as a header, pass along to the user service to verify
    e. Get /StudyLink/v1/users/ → retriever all users along with JWT and Google Token as the header

2. **Study Service**
    a. Get /StudyLink/v1/study-group/{group_id} → get request to the study-group microservice
    b. Get /StudyLink/v1/study-group/" → get a request to the study-group microservice (retrieve all groups)
    c. Post /StudyLink/v1/{user_id}/study-group/ → *asynchronous* post request to user service + study-group + perform rollback operation if the post is successful but the get request to the user service is not successful
    d. Delete /StudyLink/v1/{user_id}/study-group/{group_id} → synchronous call to the user service + study-group to delete a group
    e. Put /StudyLink/v1/{user_id}/study-group/{group_id}" → synchronous call to the user service + study-group to update a group

3. **Course Enrollment**
    a. Two get requests to retrieve a student's courses and find students of the course

4. **Chat Service**
    a. Post /StudyLink/v1/{user_id}/conversations → post a chat; has choreography pattern to call the user service and the chat service
    b. Delete "/StudyLink/v1/{user_id}/conversations/{conversation_id}", → Has google workflow to call the user and chat services
    c. Put /StudyLink/v1/conversations/{conversation_id} → Synchronous call between the user and chat service
    d. Get StudyLink/v1/{user_id}/conversations/{conversation_id} → Retrieves conversation
    e. Get /StudyLink/v1/conversations → Get all the conversations

OpenAPI: Link



Logging

```
---
insertId: 121uq80f4op40g
logName: projects/cloud-computing-437302/logs/user-login
receiveTimestamp: '2024-12-15T22:15:52.995159967Z'
resource:
  labels:
    module_id: default
    project_id: cloud-computing-437302
    version_id: 20241215t171157
    zone: projects/90605875943/zones/us6
  type: gae_app
textPayload: requesting the user logging function
timestamp: '2024-12-15T22:15:52.995159967Z'
---
insertId: 121uq80f4op40f
jsonPayload:
  logging.googleapis.com/diagnostic:
    instrumentation_source:
    - name: python
      version: 3.11.3
logName: projects/cloud-computing-437302/logs/user-login
receiveTimestamp: '2024-12-15T22:15:52.995159967Z'
resource:
  labels:
    module_id: default
    project_id: cloud-computing-437302
    version_id: 20241215t171157
    zone: projects/90605875943/zones/us6
  type: gae_app
timestamp: '2024-12-15T22:15:52.995159967Z'
---
insertId: 3doemef380s2w
logName: projects/cloud-computing-437302/logs/user-login
receiveTimestamp: '2024-12-15T22:00:34.346478387Z'
resource:
  labels:
    module_id: default
    project_id: cloud-computing-437302
    version_id: 20241215t165646
    zone: projects/90605875943/zones/us6
  type: gae_app
textPayload: requesting the user logging function
timestamp: '2024-12-15T22:00:34.346478387Z'
---
insertId: 3doemef380s2v
jsonPayload:
```

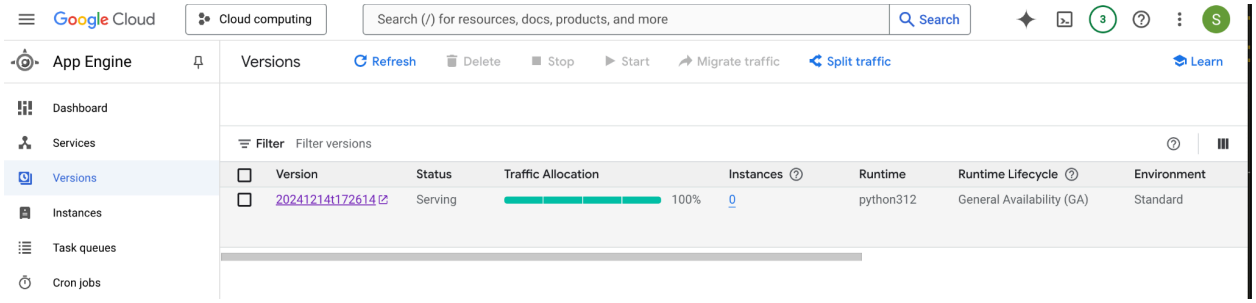
```python
    self.logging_client = logging.Client()

#this will perform the rest call
def get_user_login(self, user_id:str, google:str):
    payload = {
        "Google-Token": google
    }
    url = f"{self.user_config}/users/{user_id}/login"
    try:
        print("google",google)
        print("calling get user",google)
        log_name = "user-login"
        logger = self.logging_client.logger(log_name)
        text = "requesting the user logging function"
        logger.log_text(text)

        response = requests.get(url, headers=payload, timeout=10)
        response.raise_for_status()

        print(self.user_config, user_id)
        return (response.json(), response.status_code)
    except requests.exceptions.Timeout:
        print("The request timed out!")
        return ({"error": "The request timed out"},408)
    except HTTPError as http_err:
        print(f"HTTP error occurred: {http_err}")
        try:
        # Try to parse the response as JSON
            response_data = response.json()  # This is from the external
            print(f"External service response (JSON): {response_data}")
```

Deployment On PaaS

app.yaml:

```yaml
runtime: python312
entrypoint: gunicorn -w 4 -k uvicorn.workers.UvicornWorker app.main:app
env_variables:
  USER_MICRO : "https://9lh3sz8zwa.execute-api.us-east-1.amazonaws.com/user"
  STUDY_MICRO : "https://9lh3sz8zwa.execute-api.us-east-1.amazonaws.com/studygroup"
  CHAT_MICRO : "https://9lh3sz8zwa.execute-api.us-east-1.amazonaws.com/chat"
  COURSE_MICRO : "https://9lh3sz8zwa.execute-api.us-east-1.amazonaws.com/course"
  PROJECT: "cloud-computing-437302"
  LOCATION : "us-central1"
  WORKFLOW : "myFirstWorkflow"
```

Google deployment:



Pub-Sub

```python
import threading
from app.services.service_factory import ServiceFactory
class Publisher:
    def __init__(self):
        self.subscribers = {}

    def subscribe(self, subscriber, topic):
        print(topic, subscriber)
        if topic not in self.subscribers:
            self.subscribers[topic] = []
        self.subscribers[topic].append(subscriber)

    def publish(self, message, topic, params):
        if topic in self.subscribers:
            for subscriber in self.subscribers[topic]:
                subscriber.event.set()
                subscriber.message = message
                subscriber.params = params

class Subscriber:
    def __init__(self, name):
        self.name = name
        self.event = threading.Event()
        self.message = None
        self.params = []
        service = ServiceFactory()
        self.compo_resource = service.get_service("CompositeResource")
        self.user_error = False

    def receive(self):
        self.event.wait()
        #will sent the request to the client
        print(f"{self.name}" +"received message:" + f"{self.message}")
        if self.message == "Get User":
            response_data_get, status_code = self.compo_resource.get_user(self.params[0],self.params[1
            if  status_code != 200:
                self.event.clear()
                self.user_error = True
                return (False, response_data_get, status_code)
            else:
                self.event.clear()
```

Sync-Asynchronous

Async

```python
async def create_group(self,  user_id:str, group_data:dict, google_user:str, jwt_payload:str):
    print("staring the create group ops")
    response_get, response_post = await asyncio.gather(
        self.compo_resource.get_user_sync_internal(user_id, google_user, jwt_payload),
        self.compo_resource.create_group(group_data)
        )
    print(response_post)
    data, response_get_status =  response_get

    response_data, response_post_status=  response_post

    print(f"GET Response Status: {response_get_status}")
    print(f"POST Response Status: {response_post_status}")

# Now handle the conditions based on the status codes
    if response_get_status == 200 and response_post_status != 200:
        raise HTTPException(status_code=response_post_status, detail=response_data)

    elif response_get_status == 200 and response_post_status == 200 or response_post_status == 201:
        return {"message": response_data}

    elif response_get_status != 200 and response_post_status == 200:

        group_id = response_data.get("group_id")
        response_delete_data, status_delete = await self.compo_resource.delete_group_async(group_id)

        if status_delete != 200:
            raise HTTPException(status_code=status_delete, detail=f"rollback failed,{response_delete_data}")

        raise HTTPException(status_code=400, detail="GET request failed, rollback succedded, Post Suspended")

    elif response_get_status != 200 and (response_post_status != 200  and response_post_status !=201):
        print("Both GET and POST requests failed.")
        raise HTTPException(status_code=response_post_status, detail=response_data)
```

Sync

```python
def update_group(self,user_id:str, group_id:int, update_data:dict, google_user:str, jwt_payload:str):

    response_data_get, status_code = self.compo_resource.get_user(user_id,google_user, jwt_payload)

    if  status_code != 200:
        raise HTTPException(status_code=status_code, detail=response_data_get)

    print("update data1", update_data)
    response_data, status_code = self.compo_resource.update_group(group_id, update_data)

    if status_code != 200:

        raise HTTPException(status_code=status_code, detail=response_data)
    else:
        return response_data
```

Google Workflow

Workflow.yaml

```yaml
main:
  params: [args]
  steps:
    - callGetUserService:
        call: http.get
        args:
          url: ${args.user_url + "/users/" + args.user_id + "/profile"}
          timeout: 20
          headers:
            Google-Token: ${args.google_token}
            Authorization: ${args.authorization}
        result: getResponse

    - checkGetResponse:
        switch:
          - condition: ${getResponse.statusCode == 200}
            next: callDeleteChatService
          - condition: ${getResponse.statusCode != 200}
            next: handleError

    - callDeleteChatService:
        call: http.delete
        args:
          url: ${args.chat_url + "/conversations/"+ args.chat_id}
          headers:
            Google-Token: ${args.google_token}
            Authorization: ${args.authorization}
        result: deleteResponse

    - handleDeleteResponse:
        switch:
          - condition: ${deleteResponse.statusCode == 200 or deleteResponse.statusCode == 204}
            next: returnSuccess
          - condition: ${deleteResponse.statusCode != 200 and deleteResponse.statusCode != 204}
            next: handleError

    - returnSuccess:
        return:
          message: "Workflow completed successfully."
          get_response_body: ${getResponse.body}
          delete_response_status: ${deleteResponse.statusCode}

    - handleError:
        return:
          error: "An error occurred during the workflow."
          get_response_body: ${getResponse.body}
          delete_response_status: ${deleteResponse.statusCode}
```

Google workflow client

```python
workflows_client = workflows_v1.WorkflowsClient()
client = executions_v1.ExecutionsClient()
project = os.getenv("PROJECT")
location = os.getenv("LOCATION")
workflow = os.getenv("WORKFLOW")
parent = workflows_client.workflow_path(project, location, workflow)
print(parent)
service = ServiceFactory()
config = service.get_service("CompositeResourceService")
user_config = config.get_user_config()
chat_config = config.get_chat_config()
args = {

        "user_url": user_config,          # User service URL
        "user_id": user_id,                       # User ID
        "chat_url": chat_config,     # Chat service URL
        "chat_id": conversation_id,                       # Chat ID
        "google_token": google_user,        # Google token
        "authorization": jwt_payload   # Authorization header

}
argument_json = json.dumps(args)
execution_request = executions_v1.CreateExecutionRequest(
    parent=parent,
    execution={
        "argument": argument_json
    }
)


response = client.create_execution(request=execution_request)

print(f"Execution name: {response.name}")
print(f"Execution State: {response.state}")
execution_finished = False
backoff_delay = 1  # Start wait with delay of 1 second
print("Poll for result...")
while not execution_finished:
    execution = client.get_execution(request={"name": response.name})
    execution_finished = execution.state != executions.Execution.State.ACTIVE

    # If we haven't seen the result yet, wait a second.
    if not execution_finished:
        print("- Waiting for results...")
        time.sleep(backoff_delay)
        # Double the delay to provide exponential backoff.
        backoff_delay *= 2
    else:
        print(f"Execution finished with state: {execution.state.name}")
        print(f"Execution results: {execution.result}")
        print(f"execution error {execution.error}")
        return execution
```

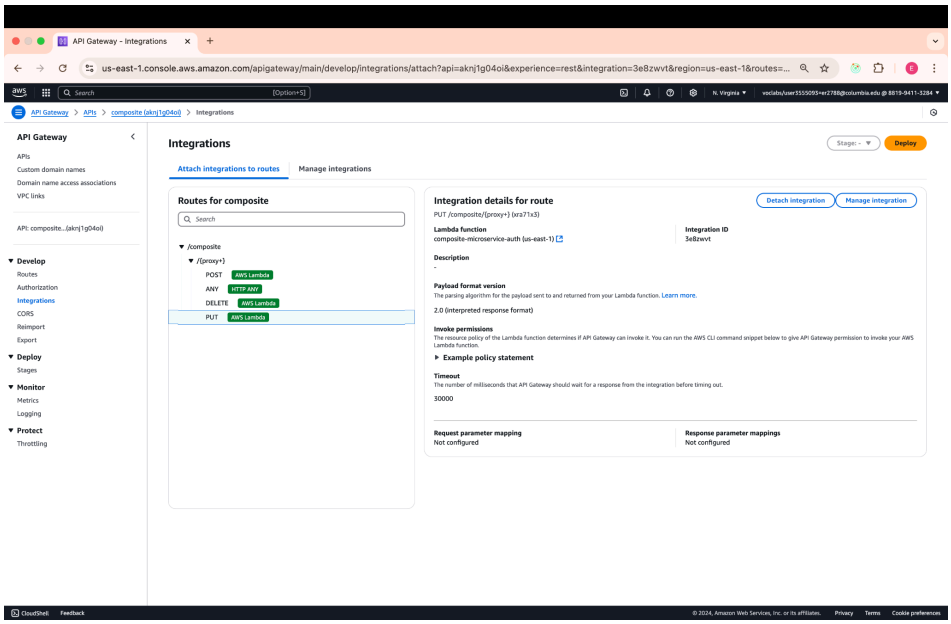
UI

The React and Redux-based UI connects with the User Microservice, enabling account creation, deletion, login, and retrieval of all user profiles. It's also integrated with the Courses Microservice, which allows users to view information on their enrolled courses. The UI is deployed on Google Cloud Storage and it is connected to the composite microservice via the API Gateway link. Information about the API Gateway Link and the Google Client ID (needed for google login) are configured in a .env file.

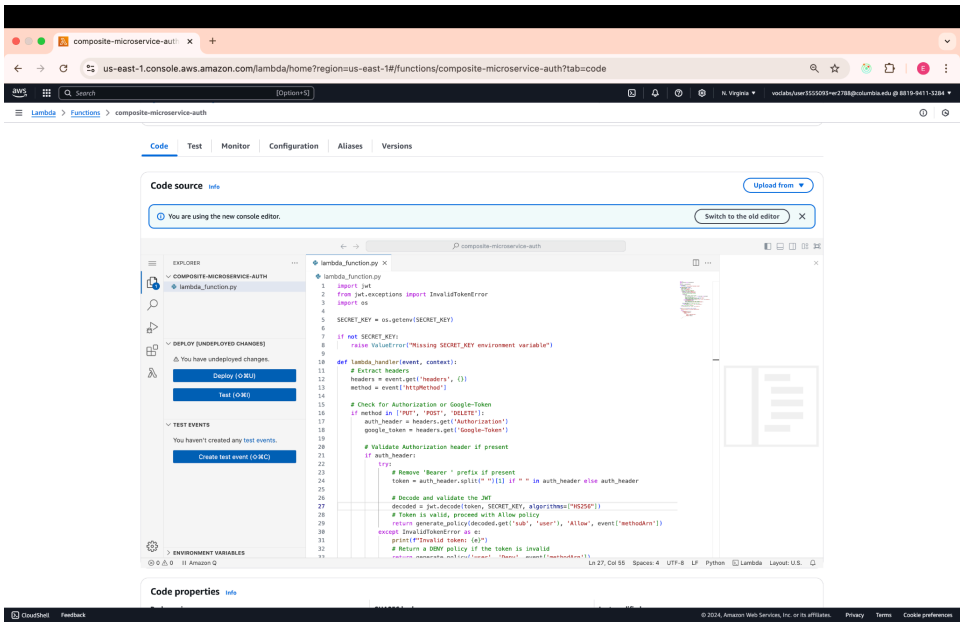
The UI allows to do the following:

- **Access/Logout:**
    - Signup - via Google Login + Courseworks Token
    - Login - via Google Login
    - Logout
    - Delete Account
- **Study Groups:**
    - Create - only for courses that the user is taking
    - Join - any group associated with a course that the user is taking
    - Edit - only for study groups created by the logged user
    - Delete - only for study groups created by the logged user
- **Calendar** - view on a calendar when the study groups that the user has joined will meet. This section relies on the StudyGroup microservice, to show calendar events for the sturdy groups that the user has joined.
- **Chat** - with users on the application, or with groups that the user has joined. This section includes integration with pagination implemented in the chat microservice.

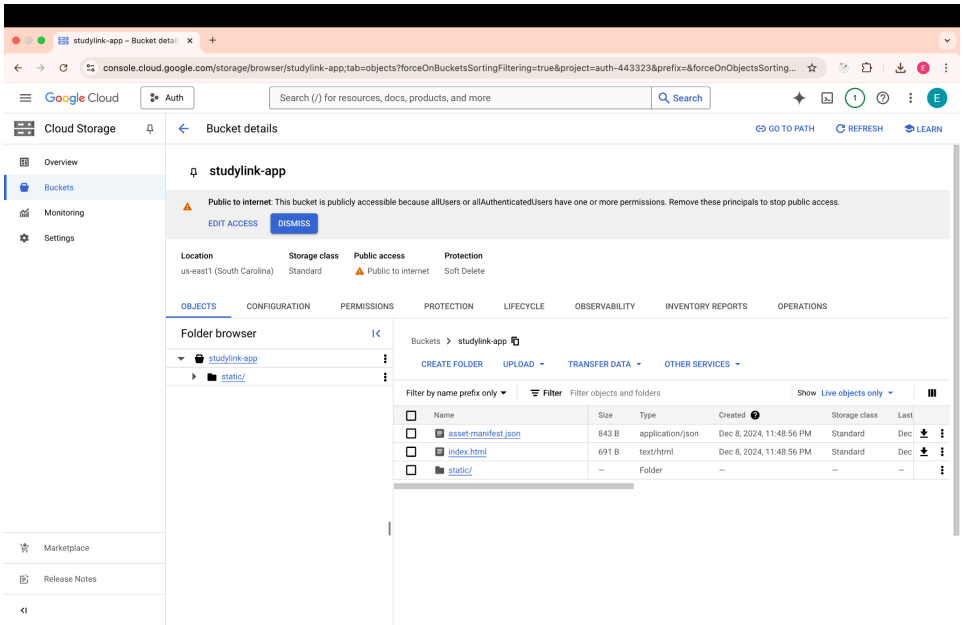
API Gateway



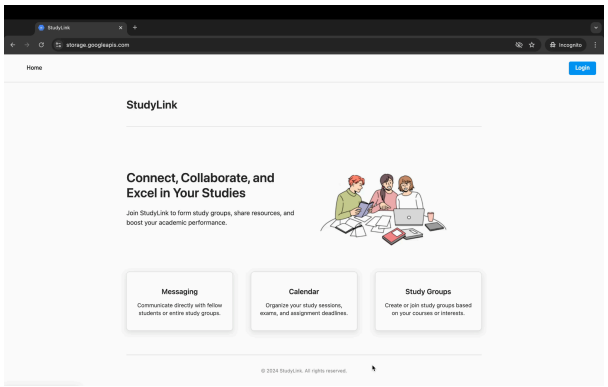
Lambda Auth (for API Gateway)



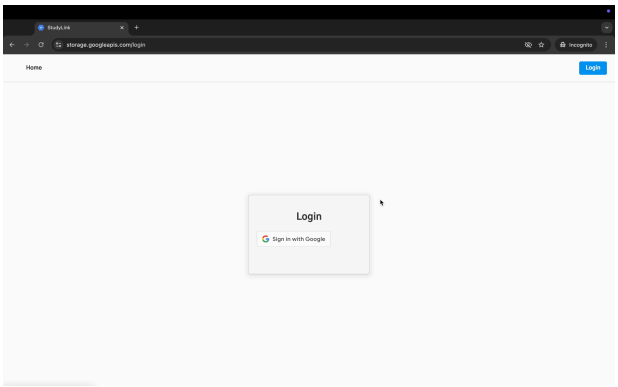
UI Hosted on Google Storage



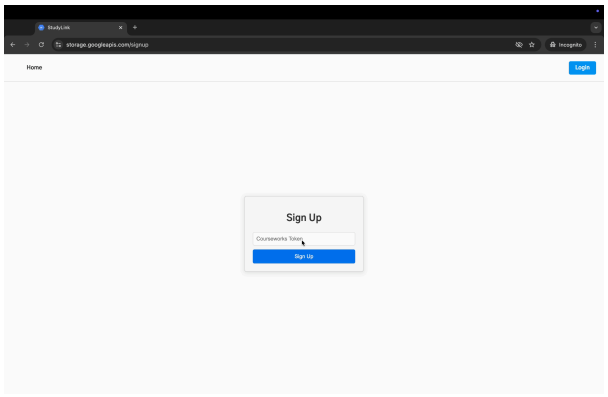
Homepage



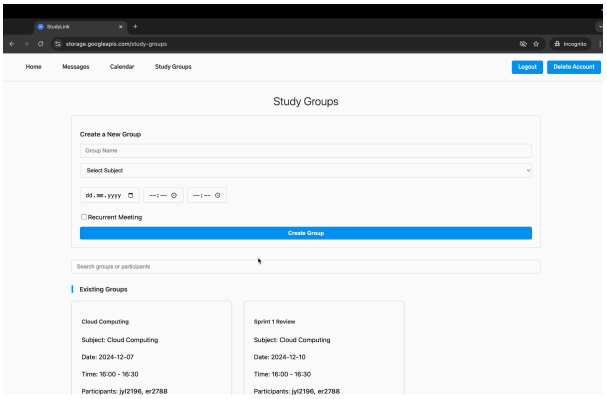
Sign In



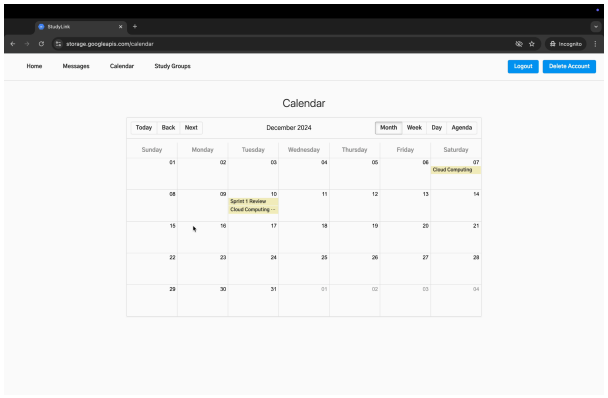
Sign Up (adding Courseworks Token)



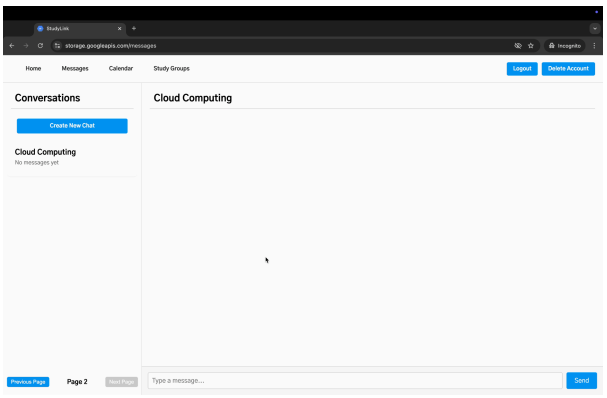
Study Group Page



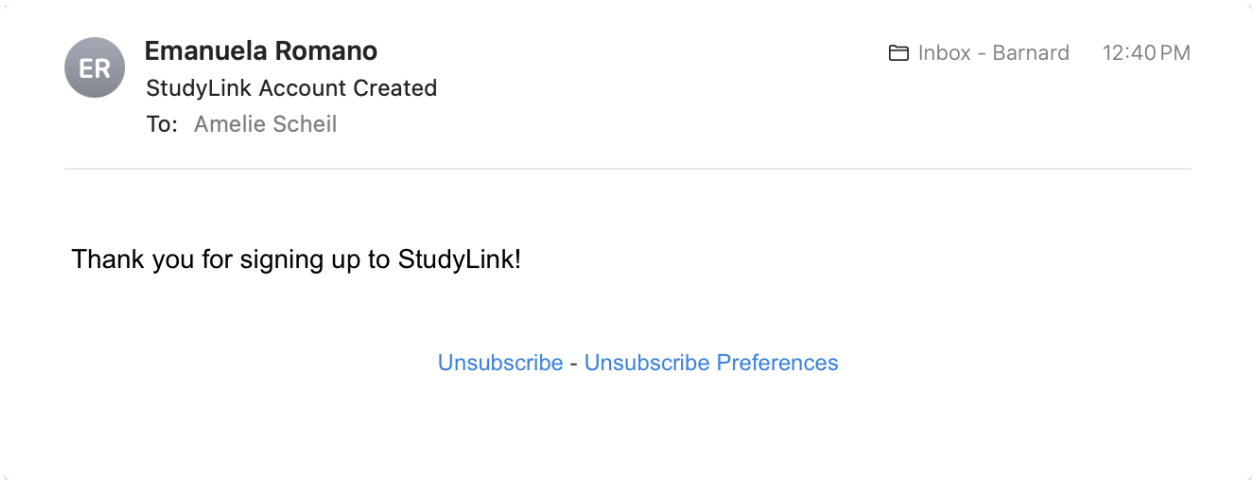
Calendar



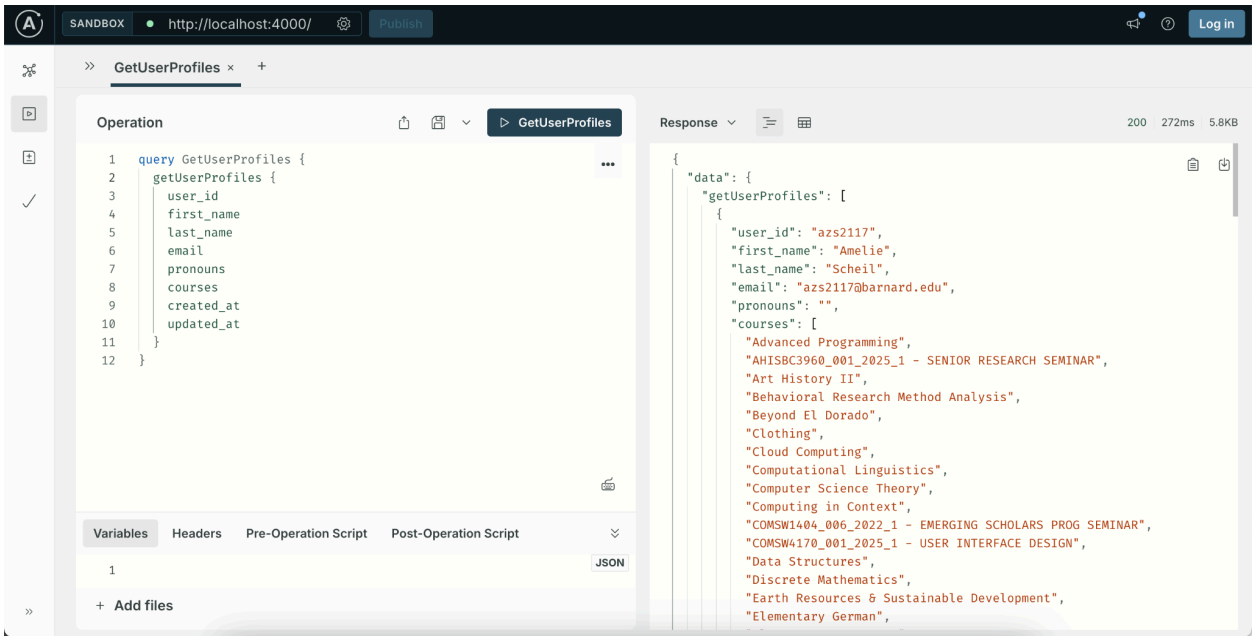
Chat (with pagination)



End User Notification



CQRS


SANDBOX
http://localhost:4000/
Publish
Log in
GetUserProfiles
Operation
GetUserProfiles
query GetUserProfiles {
getUserProfiles {
user_id
first_name
last_name
email
pronouns
courses
created_at
updated_at
}
}
Variables
Headers
Pre-Operation Script
Post-Operation Script
JSON
Add files
Response
200
272ms
5.8KB
{
"data": {
"getUserProfiles": [
{
"user_id": "azs2117",
"first_name": "Amelie",
"last_name": "Scheil",
"email": "azs2117@barnard.edu",
"pronouns": "",
"courses": [
"Advanced Programming",
"AHISBC3960_001_2025_1 - SENIOR RESEARCH SEMINAR",
"Art History II",
"Behavioral Research Method Analysis",
"Beyond El Dorado",
"Clothing",
"Cloud Computing",
"Computational Linguistics",
"Computer Science Theory",
"Computing in Context",
"COMSW1404_006_2022_1 - EMERGING SCHOLARS PROG SEMINAR",
"COMSW4170_001_2025_1 - USER INTERFACE DESIGN",
"Data Structures",
"Discrete Mathematics",
"Earth Resources & Sustainable Development",
"Elementary German",

Link to Repositories, UI Demo, and Presentation

- Presentation Slides
- UI Demo
- StudyLink Project Repo
    - Project Dashboard
- User Interface
- Microservices
    - User Microservice
    - Course Enrollment Microservice
    - Study Group Microservice
    - Chat Microservice
    - Composite Microservice
- Data Store

Contributions

Team Member	**Contribution**

Amelie	• User Microservice
	  ◦ REST functionality
	  ◦ HATEOAS links
	  ◦ Middleware logging
	  ◦ Correlation ID
	  ◦ Communication with Courseworks through API
	• CQRS
	• Domains Model

Emanuela	• User Interface • API Gateway • Elastic IPs • Google Login • JWT Tokens • FaaS • UI on Google Storage

Jeannie	• Created All Initial Github Repos
	• Created Github Projects Dashboard
	• Created Course Enrollment Microservice
	  ◦ REST functionality
	  ◦ HATEOAS links
	  ◦ Middleware logging
	  ◦ Correlation ID

	○ Communication with Courseworks through API ● Github action that deploys microservice on commit ● New Use Case diagram
Jessica	● Study groups microservice with: ○ Logging ○ Tracing ○ Correlation ID throughout REST API calls ○ Docker container on VM ● Finalized resource diagram
Jonathan	● Chat Microservice ○ Rest functionality ○ Logging/Tracing ○ Pagination ○ Asynchronous API ○ Correlation ID propagation ● Application Deployment ○ GCP Instance
Sumya	● Composite Microservice ○ Rest Functionality ○ Synchronous call ○ Async/parallel/rollback call ○ Choreography (coded) ○ OrchestrationApplication Deployment ○ Cloud Logging ● PaaS Deployment