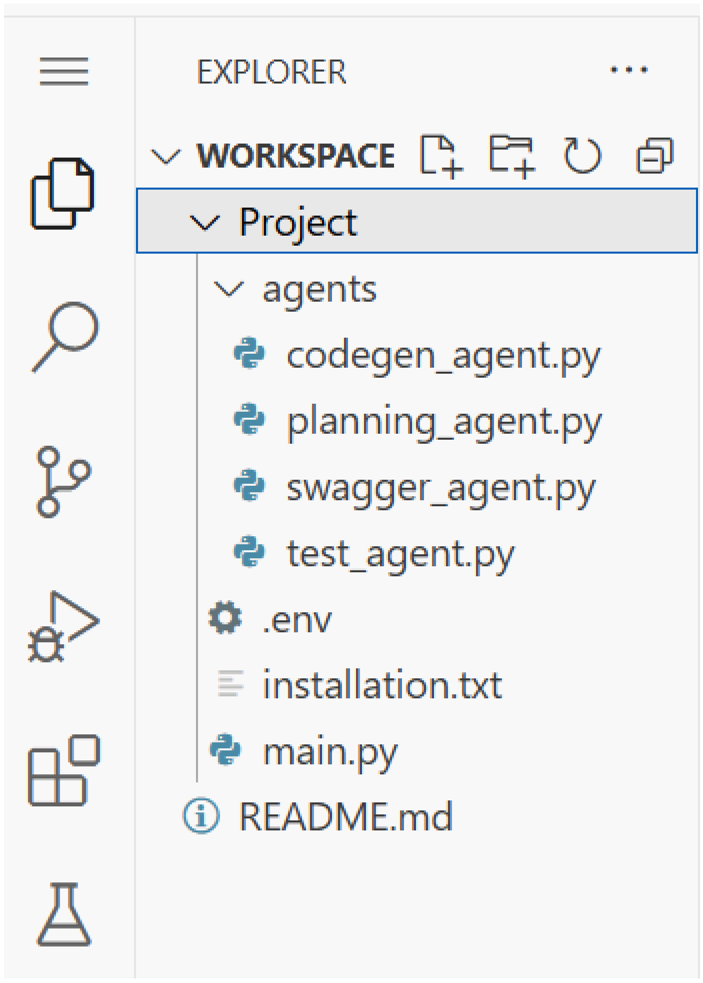
CodeCrafter AI: GenAI-Powered Microservice Builder

Overview
Microservices architecture offers scalability, flexibility, and independent deployment for complex software systems. In this hackathon, your challenge is to build a Generative AI-powered Microservice Code Generator that takes a natural language user story as input and outputs:
Microservice architecture recommendation
Service-wise controller and business logic code
Swagger/OpenAPI documentation
Unit test code for each service
You can use any LLM for generating and reasoning about code and documentation and build the system using Python and Streamlit.
Objective
Given a user story and selected backend language (e.g., Java, Node.js, or .NET), your system should:
Decompose the story into features and services.
Recommend a microservice-based architecture.
Generate controller and service code per service.
Create Swagger documentation for the APIs.
Generate unit test code for each service.
All generated files must be stored in a structured output directory.
File Structure:



```
EXPLORER
WORKSPACE
  Project
    agents
      codegen_agent.py
      planning_agent.py
      swagger_agent.py
      test_agent.py
    .env
    installation.txt
    main.py
  README.md
```

Given:
installation.txt: The packages to be installed.

Modules to Implement:
Note: Please use multiple API keys if using any free tier llms via APIs.
1) planning_agent.py – User Story Analyzer & Architecture Planner
Purpose: This module acts as the entry point in the microservice generation pipeline. It uses llm to analyze a natural language user story and extract actionable technical components that inform the system architecture and service decomposition.
Function: run_planning_agent(user_story: str, language: str) → dict
Responsibilities:
Feature Extraction
Identifies core functional verbs such as register, place order, login, etc.
Maps each feature to a service following standard microservice naming conventions.
Architecture Preference Identification
Scans the user story for any hints about desired architecture, database, messaging systems, or caching mechanisms. Examples: "Use Kafka for event communication", "Store user data in MongoDB".
Architecture Recommendation
Suggests a complete architecture stack (REST, Event-driven, etc.) based on:
Extracted features
Language preference
User hints (if any)

2) codegen_agent.py – Microservice Code Generator
Purpose: This module is responsible for generating backend microservice code for each identified service. It uses the llm model to produce controller logic, service (business logic), and model/schema definitions based on the provided feature set and architecture.
Function: run_codegen_agent(features: list, services: list, arch_config: dict, language: str) → dict
Responsibilities:
Iterative Code Generation
For every (feature, service) pair, constructs a prompt to Gemini including:
Feature name (e.g., "Register a new user")
Service name (e.g., UserService)
Chosen architecture (e.g., REST, gRPC)
Infrastructure stack (DB, messaging, cache)
Target language (e.g., Java, NodeJS)
Expected Code Components
Gemini returns a JSON structure with:
controller_filename and controller_code
service_filename and service_code
model_filename and model_code
Aggregated Output
All generated code is stored in a dict indexed by service name.
Format allows downstream writing into service-specific folders.

3) swagger_agent.py – Swagger (OpenAPI) Generator
Purpose: This module converts generated controller code into a valid Swagger (OpenAPI 3.0) specification using llm. It ensures that every microservice is paired with clean, machine-readable API documentation that adheres to industry standards.
Function: run_swagger_agent(service_code_map: dict, language: str) → dict
Responsibilities:

Controller-Based Documentation
Iterates through the code generated for each service.
Extracts controller_code and feeds it to Gemini along with the service name and programming language.
Instructs Gemini to return a valid OpenAPI YAML spec in a specific JSON structure.
Expected Output Format
Gemini returns a Swagger spec embedded in a JSON object.

4) test_agent.py – Unit Test Code Generator
Purpose:
This module automates the generation of unit test code for each microservice by leveraging llm. It analyzes the previously generated controller and service logic to create relevant test cases using the testing framework appropriate for the selected programming language.
Function: run_test_agent(service_code_map: dict, language: str) → dict
Responsibilities:
Test Code Generation
For each microservice:
Extracts controller_code and service_code
Uses Gemini to generate a structured test file
The test is tailored to the language's standard test framework (e.g., JUnit for Java, unittest for Python)
Expected Output Format
Gemini responds with test code wrapped in a JSON structure.

main.py – Streamlit UI & Agent Orchestration
Purpose:
The main.py module serves as the central orchestrator and user interface for the GenAI-powered microservice generator. It captures user input via a web UI built with Streamlit, invokes all backend agent modules in sequence, and displays the output with appropriate visual formatting.
Responsibilities
1. User Interface (Streamlit)
Collects:
A natural language user story
A backend language (e.g., Java, NodeJS, Python)
Displays:
Extracted features and services
Architecture recommendations
Generated controller/service/model code
Swagger API documentation
Unit test code per service
Uses expanders and columns for clean, interactive visualization.
2. Pipeline Execution
Invokes the following agents in order when the user submits input:
run_planning_agent(user_story, language)
run_codegen_agent(features, services, arch_config, language)
run_swagger_agent(service_outputs, language)
run_test_agent(service_outputs, language)
3. Output Management
Converts the user story to a safe folder name.
Organizes the output under the structure:

```
output/{story_slug}/
|── {ServiceName}/
| ├── Controller file
```

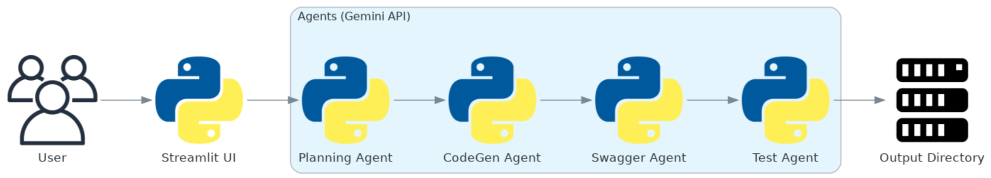
| ├── Service file
| └── Model file
├── swagger/{ServiceName}/swagger.yaml
├── tests/{ServiceName}/{test_file}

Example Use Case:

User inputs:

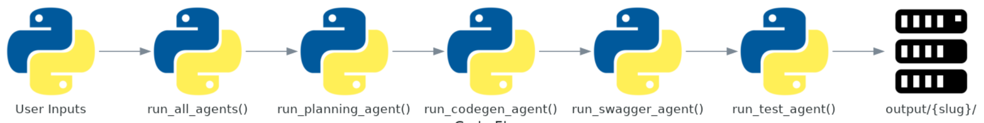
Story: "Users should register, login, and view their profile."
Language: Java
Output:
Features: ["register", "login", "view profile"]
Services: ["UserService", "ProfileService"]
Directory: output/users_should_register_login_and_view_their/
Auto-generated:
Java code files for each service
Swagger YAML documentation
Unit test class files

Architecture:



Architecture

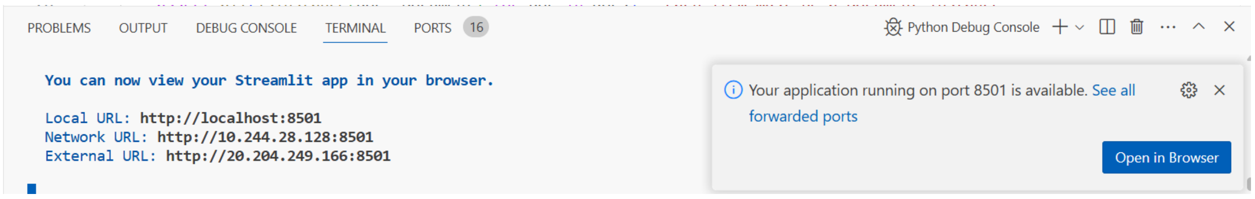
Code_flow:



Code Flow

Commands to Create a Google Gemini API Key

Open your web browser.
Launch any browser (e.g., Chrome, Firefox) on your computer.
Go to Google AI Studio.
In the address bar, type aistudio.google.com and press Enter.
Sign in to your Google account.
Click the "Sign In" button in the top-right corner.
Enter your Google email and password, then click "Next" to log in.
If you don't have an account, click "Create Account" and follow the prompts to make one.
Navigate to the API Key section.
On the Google AI Studio homepage, look at the left sidebar.
Click on "Get API Key" (usually near the top-left corner).
Create a new API key.
In the API Key section, click the "Create API Key" button.
A pop-up will appear—select "Create API Key in new project" (or choose an existing project if you have one).
Click "Create" to generate the key.
Copy the generated API key.
Once the key is created, it will appear on the screen.
Click the "Copy" button next to the key (or highlight it and press Ctrl+C/Command+C).
Save the key in a secure place (e.g., a text file or password manager) because it won't be shown again.
Implementation Explanation:
Before executing the main.py, enter the **Gemini API key** in the **.env** file.
Open the **main.py** integrated terminal.
check the path it in the Project directory, if not use **cd** command to navigate.
To install required packages, run **python3 main.py** in terminal or click the run & debug button for main.py
Use **python3 -m streamlit run main.py** to execute the application, then you will get the pop-up window below, click the assigned port (**Open in Browser**) which will navigate to streamlit application window.



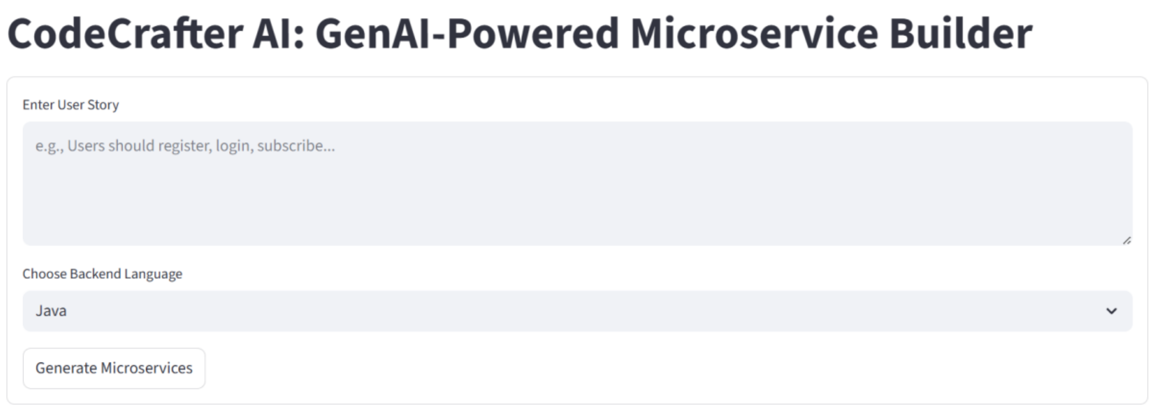
```
You can now view your Streamlit app in your browser.

Local URL: http://localhost:8501
Network URL: http://10.244.28.128:8501
External URL: http://20.204.249.166:8501
```

Your application running on port 8501 is available. See all forwarded ports

IMPORTANT: Before you are running testcases make sure your Gemini API free tier is not exhausted.
To check the testcases, you can use **python3 -W ignore -m pytest tests.py -v** (check the directory it should be **Project** directory)
Sample Output:

The below shared image format must be the first page of display once the streamlit starts running.


CodeCrafter AI: GenAI-Powered Microservice Builder

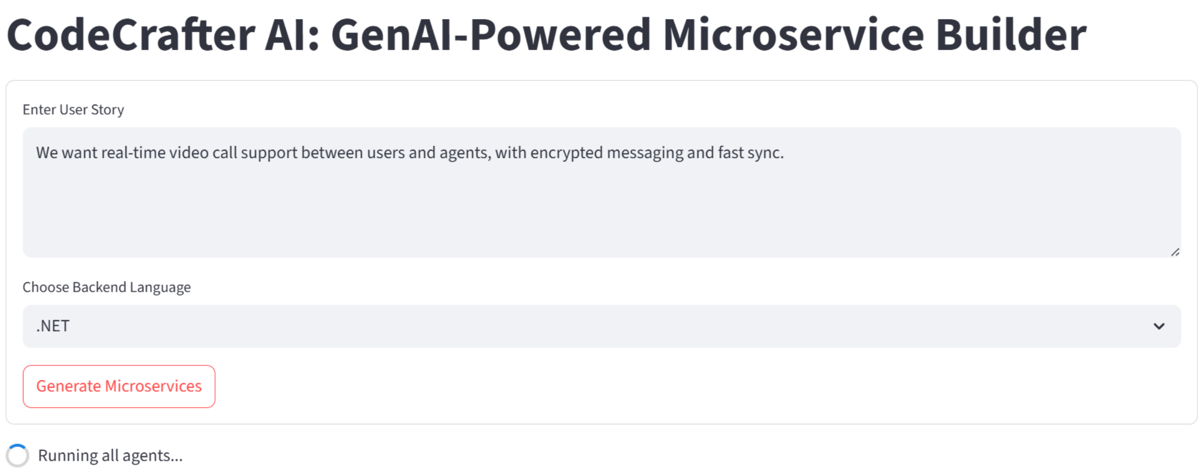
Enter User Story

e.g., Users should register, login, subscribe...

Choose Backend Language

Java

Enter the user-story and select a backend language ("Java", "NodeJS", ".NET", "Python", "Go", "Ruby", "PHP", "Kotlin").


CodeCrafter AI: GenAI-Powered Microservice Builder

Enter User Story

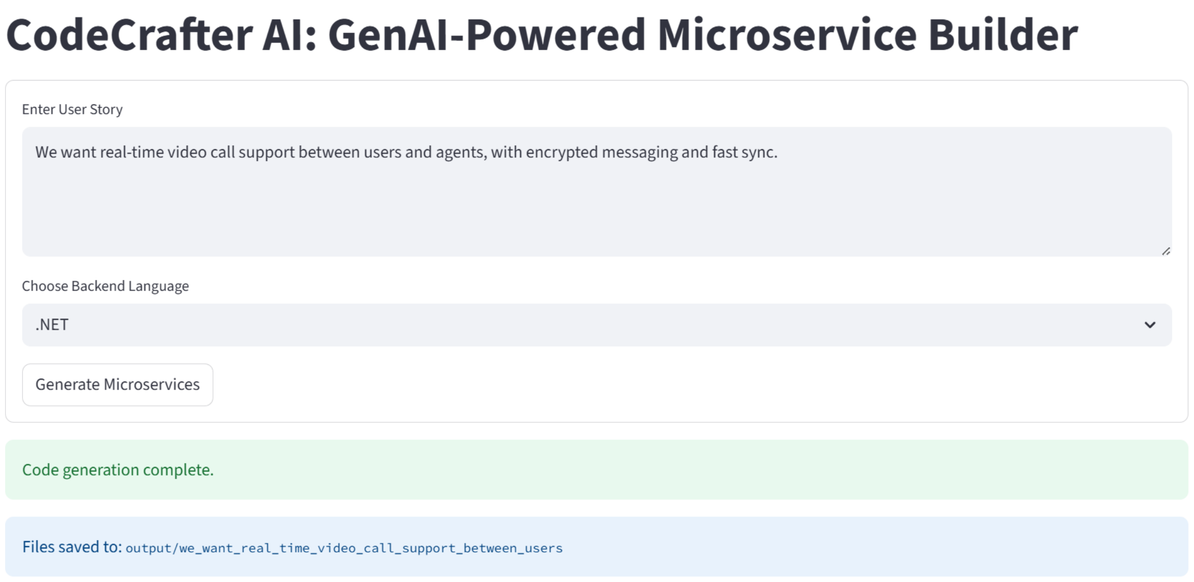
We want real-time video call support between users and agents, with encrypted messaging and fast sync.

Choose Backend Language

.NET

Running all agents...

Once filled and selected "Genarate Microservices" button, we need to do the back-end processing



CodeCrafter AI: GenAI-Powered Microservice Builder

Enter User Story

We want real-time video call support between users and agents, with encrypted messaging and fast sync.

Choose Backend Language

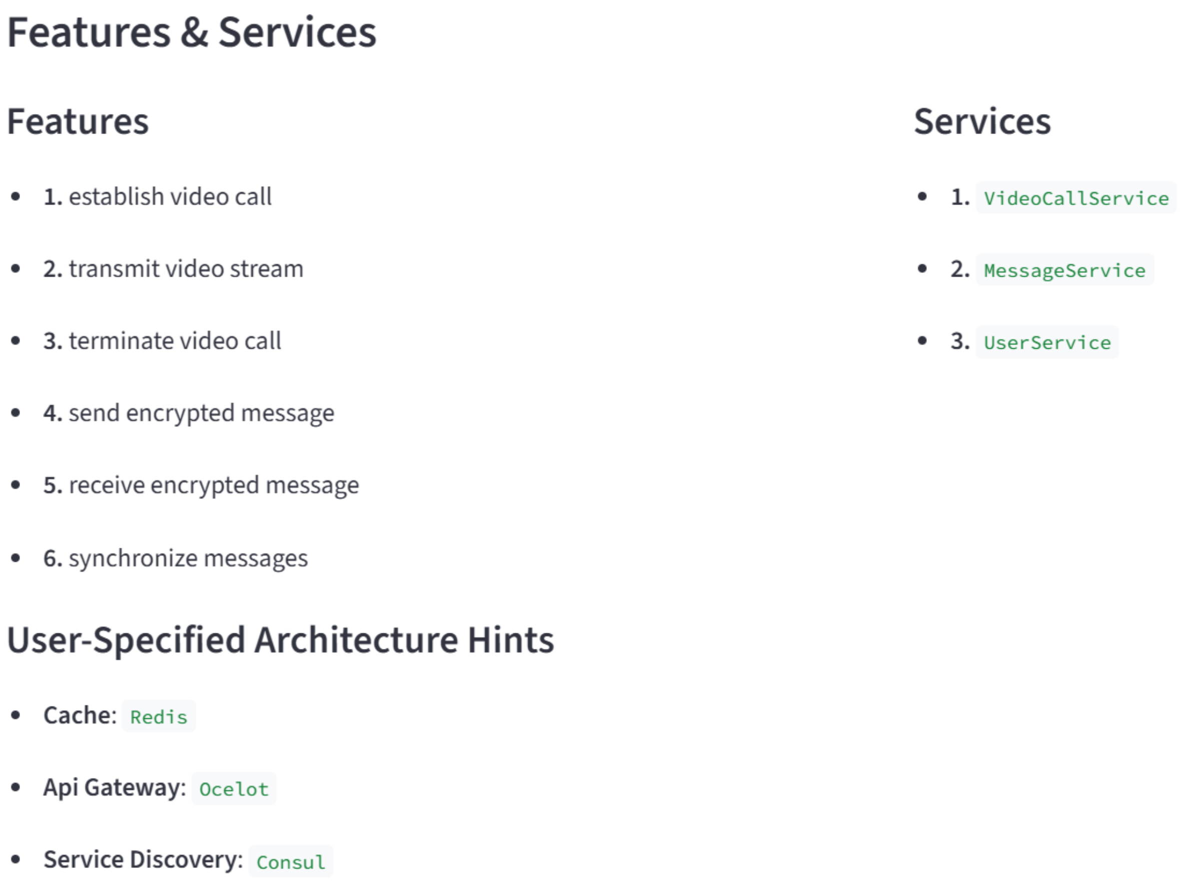
.NET

Generate Microservices

Code generation complete.

Files saved to: `output/we_want_real_time_video_call_support_between_users`

We need to display the extracted features & services.



Features & Services

Features

- **1.** establish video call
- **2.** transmit video stream
- **3.** terminate video call
- **4.** send encrypted message
- **5.** receive encrypted message
- **6.** synchronize messages

Services

- **1.** `VideoCallService`
- **2.** `MessageService`
- **3.** `UserService`

User-Specified Architecture Hints

- **Cache:** `Redis`
- **Api Gateway:** `Ocelot`
- **Service Discovery:** `Consul`

Architecture Configuration

- **Architecture:** `Microservices`
- **Database:** `PostgreSQL`
- **Messaging:** `RabbitMQ`
- **Cache:** `Redis`
- **API Gateway:** `Ocelot`
- **Service Discovery:** `Consul`

Generated Code Preview

VideoCallService

MessageService

UserService

Swagger YAMLs

VideoCallService Swagger

MessageService Swagger

UserService Swagger

Unit Tests

VideoCallService Test

MessageService Test

UserService Test

Generated Code Preview

VideoCallService

```
using Microsoft.AspNetCore.Mvc;
using VideoCallService.Models;
using VideoCallService.Services;
using Microsoft.Extensions.Logging;
using System;
using System.Threading.Tasks;

namespace VideoCallService.Controllers
{
    [ApiController]
    [Route("[controller]")]
    public class VideoCallController : ControllerBase
    {
        private readonly IVideoCallService _videoCallService;
        private readonly ILogger<VideoCallController> _logger;

        public VideoCallController(IVideoCallService videoCallService, ILogger<VideoCallController> logger)
        {
            _videoCallService = videoCallService ?? throw new ArgumentNullException(nameof(videoCallService));
            _logger = logger ?? throw new ArgumentNullException(nameof(logger));
        }
```

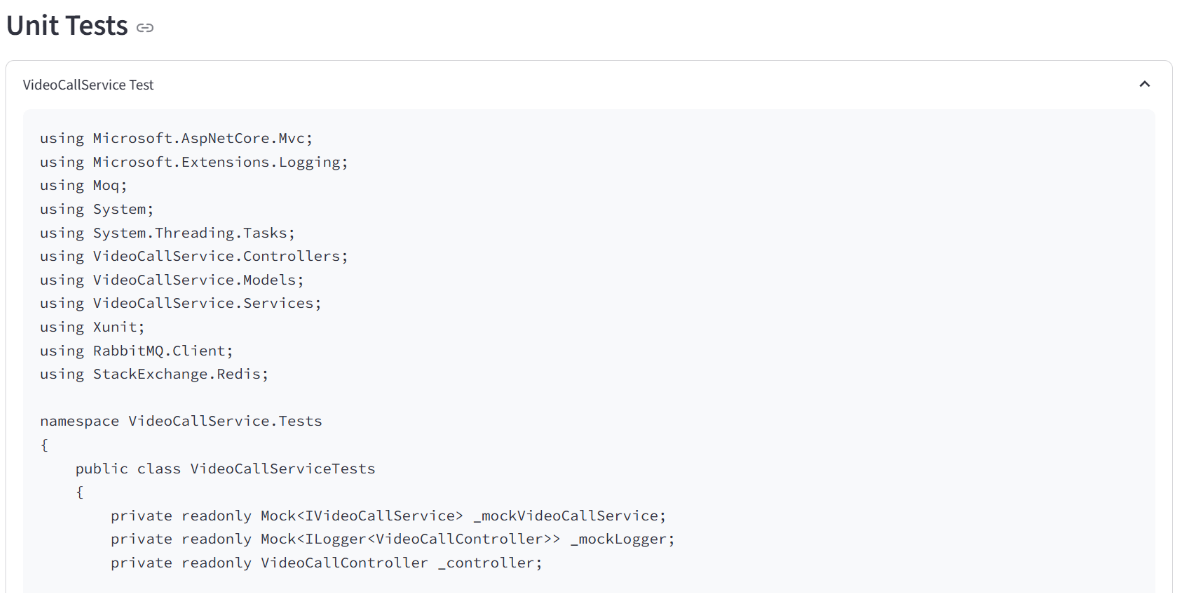
Swagger YAMLs

VideoCallService Swagger

```
---
openapi: 3.0.0
info:
  title: VideoCallService API
  version: v1

servers:
  - url: /
    description: Default server

paths:
  /VideoCall/initiate:
    post:
      summary: Initiates a video call.
      description: Creates a new video call session.
      operationId: InitiateVideoCall
      requestBody:
        required: true
```



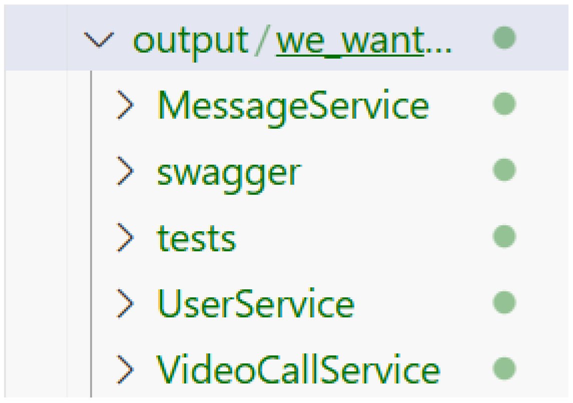
Unit Tests

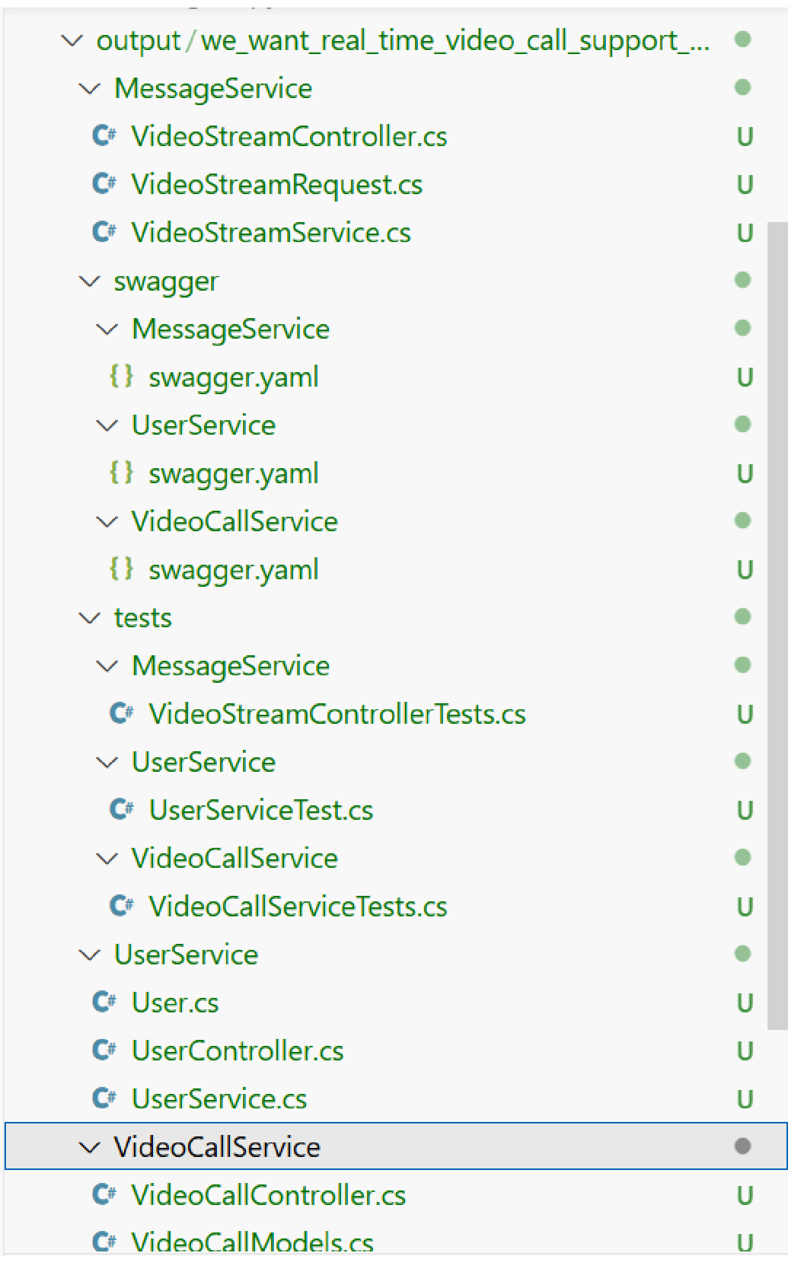
VideoCallService Test

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using Moq;
using System;
using System.Threading.Tasks;
using VideoCallService.Controllers;
using VideoCallService.Models;
using VideoCallService.Services;
using Xunit;
using RabbitMQ.Client;
using StackExchange.Redis;

namespace VideoCallService.Tests
{
    public class VideoCallServiceTests
    {
        private readonly Mock<IVideoCallService> _mockVideoCallService;
        private readonly Mock<ILogger<VideoCallController>> _mockLogger;
        private readonly VideoCallController _controller;
```

The generated code must be stored in the output directory as below.





```
output / we_want_real_time_video_call_support_...
  MessageService
    VideoStreamController.cs          U
    VideoStreamRequest.cs             U
    VideoStreamService.cs             U
  swagger
    MessageService
      swagger.yaml                    U
    UserService
      swagger.yaml                    U
    VideoCallService
      swagger.yaml                    U
  tests
    MessageService
      VideoStreamControllerTests.cs   U
    UserService
      UserServiceTest.cs              U
    VideoCallService
      VideoCallServiceTests.cs        U
  UserService
    User.cs                           U
    UserController.cs                 U
    UserService.cs                    U
  VideoCallService
    VideoCallController.cs            U
    VideoCallModels.cs                U
```