Low-Code/No-Code Backend Code Generation Engine Documentation

Dhiraj Patra

Version 1.0

1. **Introduction**

Overview of Low-Code/No-Code Backend Code Generation Engine:

The LCNC (Low-Code/No-Code) application development initiative originated from the need for a user-friendly platform that empowers individuals with limited technical expertise to create business applications.

However, the project's scope expanded to cater to freelance developers as well, enabling them to rapidly develop applications without writing code. This approach allows for initial prototype development, followed by manual optimization and scaling for enhanced functionality and performance.

Purpose of the module:

The Code Engine is a core component of the Low-Code/No-Code (LCNC) application, responsible for generating robust back-end code for applications developed within the platform.

This module automatically creates the entire back-end folder structure and required python files, streamlining development. By automating the generation of complex back-end REST servers, the Code Engine:

Saves significant development time (days/weeks) for back-end teams.

Reduces costs associated with manual development.

Minimises the need for specialised expertise.

After automatic back-end generation, manual verification ensures quality and accuracy. This foundation enables further optimization, customization and enhancement, empowering developers to refine and scale their applications efficiently.

In the context of an LCNC platform, generating the API from the frontend code is generally the more practical choice. Here’s why:

1. Frontend-Driven Approach:

- Synchronizes with UI: Since the frontend dictates the user interface and page interactions, generating the backend API based on this ensures the API matches the client’s needs, streamlining both data handling and interaction flow.

- Reduced Complexity: With a focus on UI, this approach simplifies mapping frontend components to backend services. It enables generating only the endpoints needed for user interactions, reducing unnecessary complexity in the API layer.

- Automated Updates: Any changes in the frontend can automatically trigger updates in the backend API, ensuring a responsive and adaptable codebase.

2. Backend-Driven Approach:

- API Stability: Starting from the backend can offer a more stable, well-defined API but may require frontend updates to adapt to backend changes, leading to maintenance overhead.

- Structure Complexity: If the backend drives the API, the system may create APIs that aren't directly aligned with frontend workflows, adding unnecessary load on frontend logic.

So, for LCNC platforms where user-centric flexibility is key, a frontend-driven API generation aligns better with modularity and adaptability needs.

```
example_swagger: 3.0.0
info:
  title: User Authentication API
  description: API for user registration and login.
  version: 1.0.0

paths:
  /api/v1/auth/register:
    post:
      summary: Register a new user.
      description: Creates a new user account.
      requestBody:
        required: true
        content:
          application/json:
            schema:
```

```
          type: object
          required:
            - username
            - email
            - password
            - confirmPassword
          properties:
            username:
              type: string
            email:
              type: string
            password:
              type: string
            confirmPassword:
              type: string
    responses:
      201:
        description: User created successfully.
      400:
        description: Invalid request.
      409:
        description: Username or email already exists.

/api/v1/auth/login:
  post:
    summary: Login an existing user.
    description: Authenticates an existing user.
    requestBody:
```

```
      required: true
      content:
        application/json:
          schema:
            type: object
            required:
              - username
              - password
            properties:
              username:
                type: string
              password:
                type: string
    responses:
      200:
        description: Login successful.
      401:
        description: Invalid credentials.
      404:
        description: User not found.

components:
  securitySchemes:
    bearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
```

Target audience and stakeholders:

This document addresses managerial and technical audiences. To maximise its value, readers should possess foundational knowledge of cutting-edge technologies like Generative AI and RAG, facilitating informed engagement with the module's complexities.

2. **Architecture Documentation**

2.1. **System Components**

This is the overall diagram for the Code Engine. Which will be called from a worker process. That worker process will be running on the server. When that worker reads the message from the message broker it will call the Code Engine to generate back end code.

Input for the Code Engine to generate back end code are API details in swagger YAML file. Which are front end APIs for that specific application.

Main technologies and steps used

LCNC Application Architecture Overview

Our Low-Code/No-Code (LCNC) application employs a microservices architecture, comprising multiple independent services. These services are primarily containerized using Docker, with a shared PostgreSQL database. Although microservices are inherently distributed, our initial design simplifies the structure to accommodate a small team. However, we plan to evolve our architecture by allocating separate databases for each service and leveraging message brokers for inter-service communication – an approach we've already begun implementing.

Code Engine Module: Automating Backend Structure Generation

The Code Engine module plays a pivotal role in our LCNC application. Upon completing frontend customization using UI Builders, users automatically generate Swagger API definitions in YAML format for all requisite page APIs. This YAML content is then published to a message broker.

A continuously running backend worker process, subscribed to the same message broker topic, consumes this YAML content. After processing, it

saves the data as front.yaml within a designated folder (e.g., swagger_files), which can be hosted on cloud storage solutions like Amazon S3.

Subsequently, the worker invokes the Code Engine API to generate the comprehensive backend structure, including necessary folders and files, within another specified folder (potentially hosted on a separate S3 bucket). This automated process streamlines backend development.

Deployment via DevOps and Containerization

To facilitate seamless deployment, our DevOps team is developing an additional worker. This worker replicates the generated backend folder into a designated container, enabling execution for the specific project. This containerization ensures efficient, isolated, and scalable project deployments.

Technology & Plugins

Frontend: Swagger file with all API details (e.g., registration, login pages)

OpenAI API: Integration for code generation

Prompt Template: Template for generating backend code

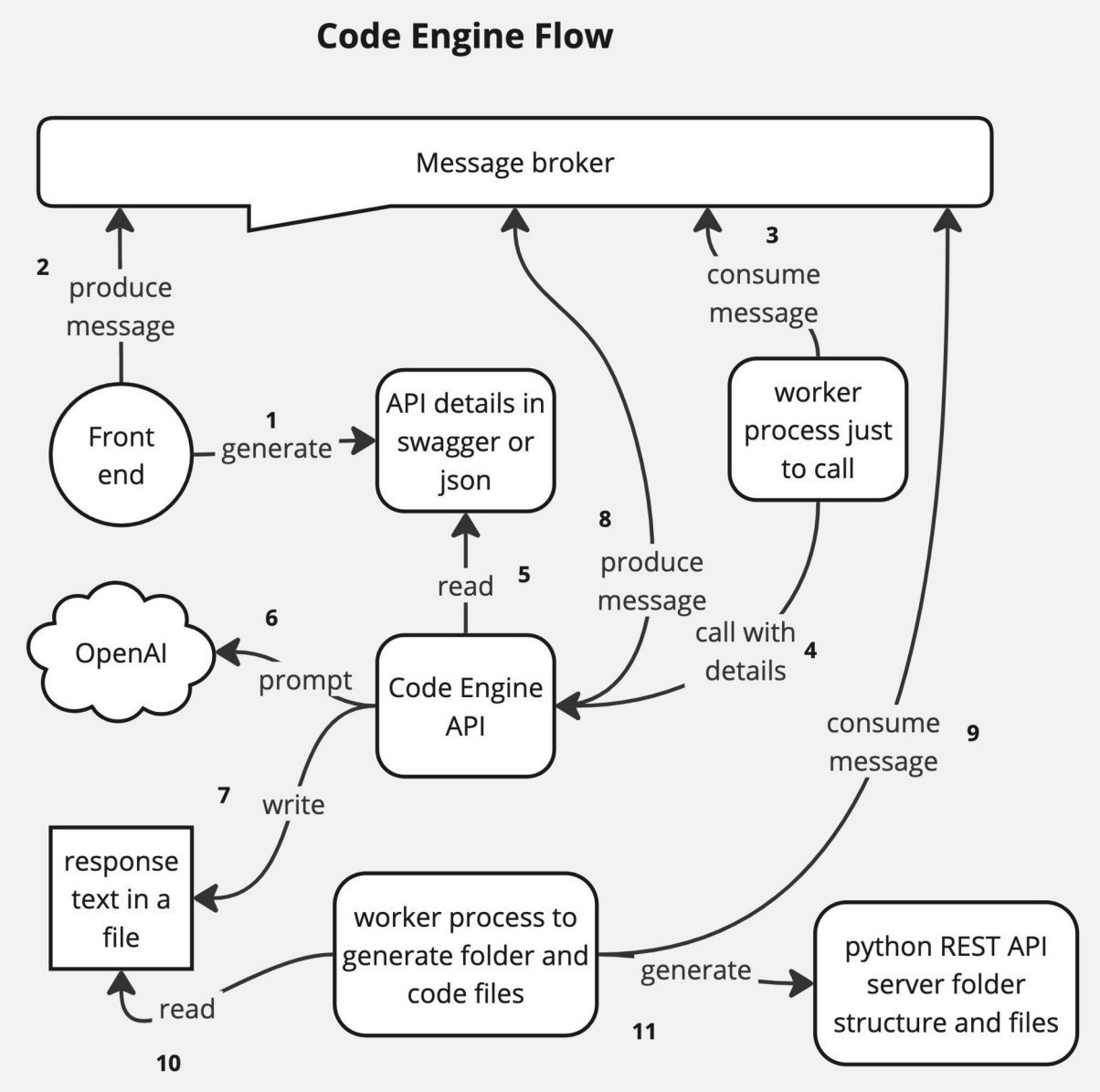
Langchain: AI-powered language processing

FastAPI: Python framework for backend code generation

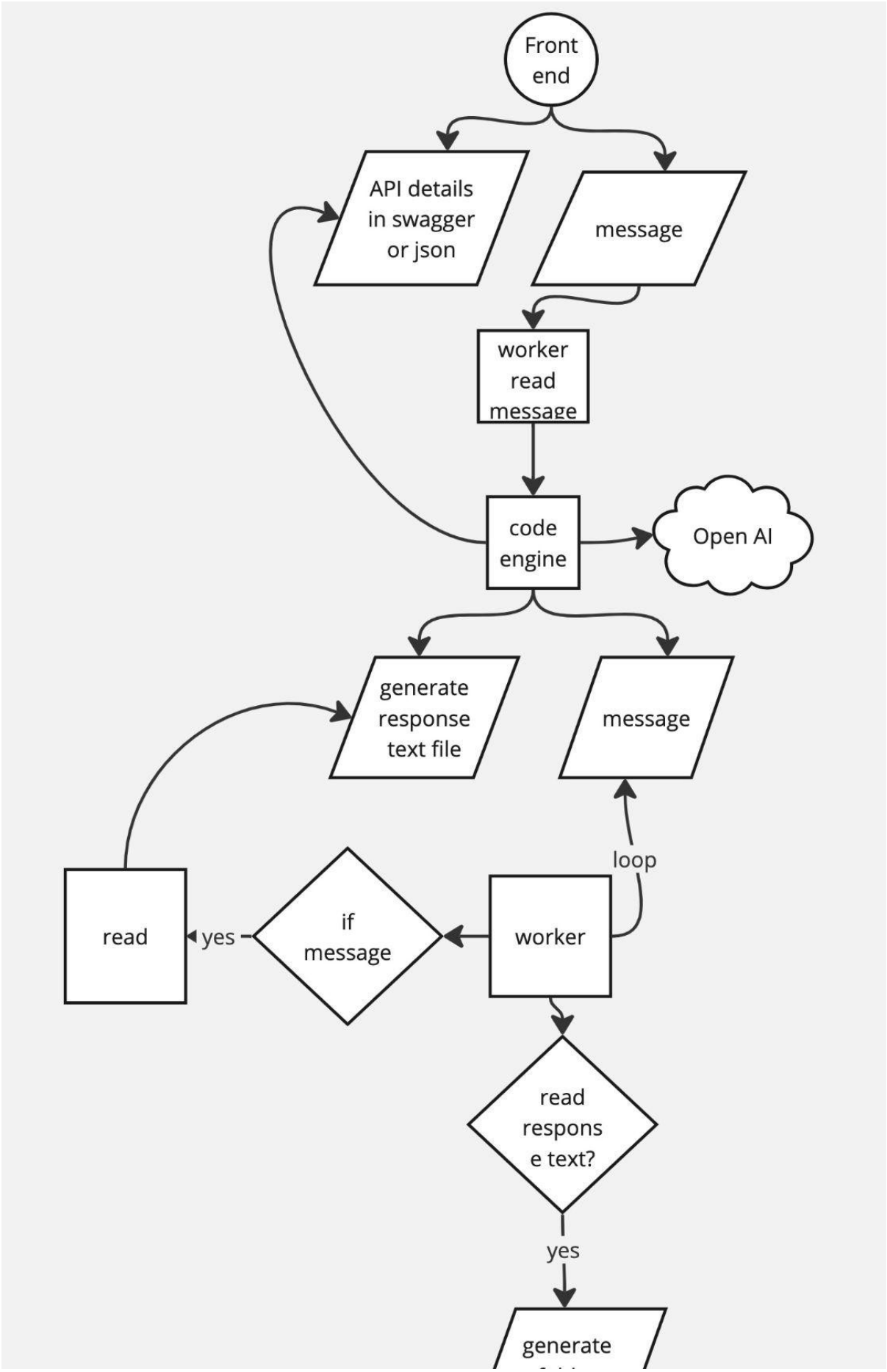
RabbitMQ: Message broker to communicate within workers and main process

2.2. **Process Flow Diagrams**

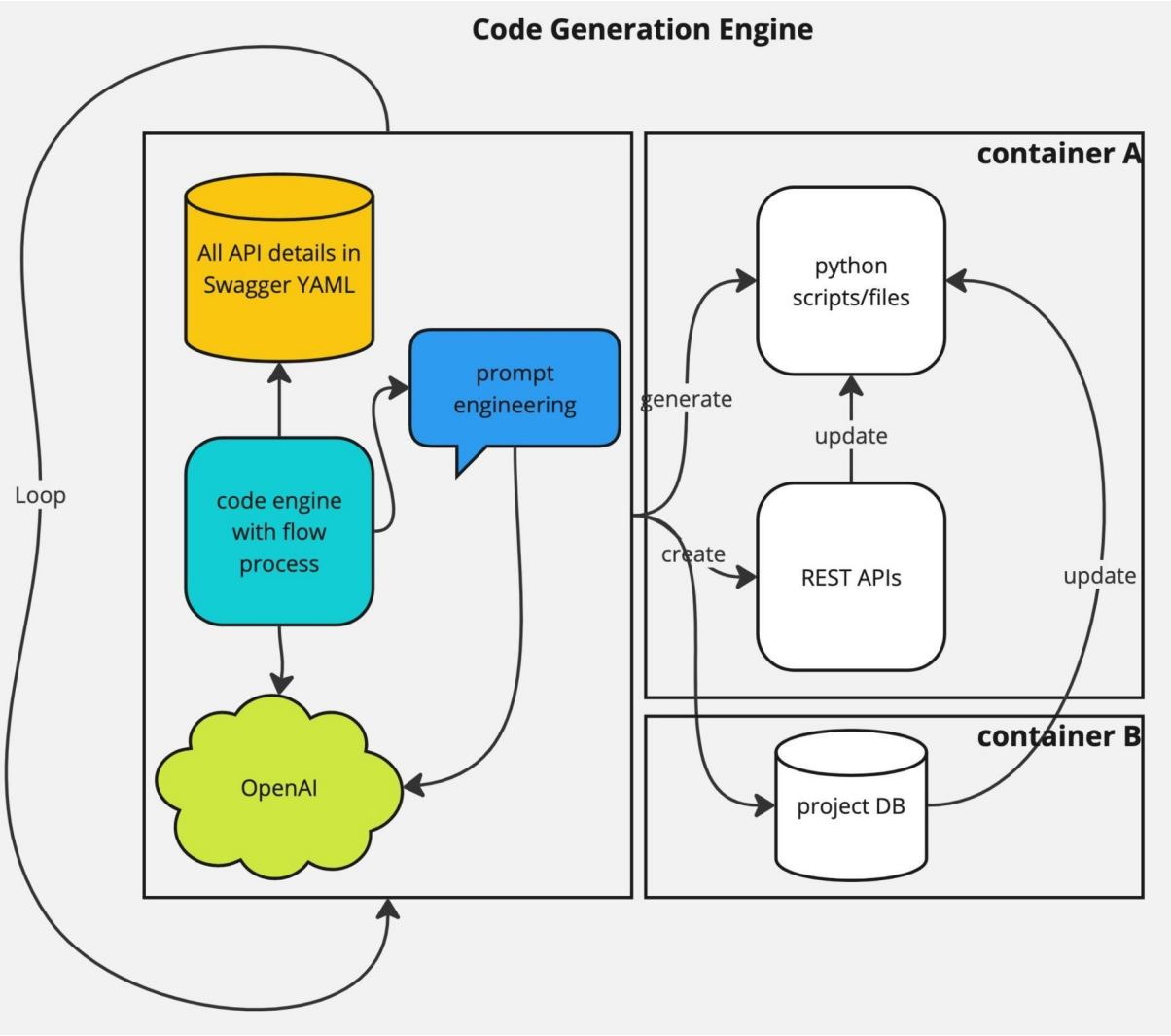
Sequence Diagram: Frontend to OpenAI API interaction



Data Flow Diagram: Data exchange between components


Front end
API details in swagger or json
message
worker read message
code engine
Open AI
generate response text file
message
loop
read
yes
if message
worker
read respons e text?
yes
generate

System Architecture Diagram: Overall system design



3. **Roadmap with Key Milestones**

Research and Planning: OpenAI API, Langchain and FastAPI integration in Docker based microservices environment. Common containers are being used eg. pgsql database container.

Prototype Development: Initial backend code generation with API swagger yaml file has completed successfully. It is now generating not only the whole REST API folder structure as well as all python back end files including the main.py which serve as main rest api services. Requirements.txt file and all other related files eg. schemas, models, database etc.

Testing and Refining: Iterative testing and improvement. Main architect and developer has tested several times in a docker container based microservices environment running on a local server.

Currently we have distributed different module development with docker containers. However before deployment we need to merge them and put all the containers in one server for production with help of DevOps. That time some common containers eg. Databases need to generalise and be the same as different containers and services using it.

Deployment: Production-ready deployment in main microservices environment server.

Maintenance and Updates: Ongoing enhancements and optimization of the whole module required. Many times OpenAI API changes their response text as well.

4. **Estimated Timelines for Each Phase**

Considering that there are fully dedicated at least six senior developers, one architect, project manager and other supporting teams available full time for this project from the beginning.

Research and Planning: 4-6 weeks

Prototype Development: 8-12 weeks

Testing and Refining: 4-8 weeks

Deployment: 2-4 weeks with experienced DevOps team for cloud

Maintenance and Updates: Ongoing and iterative process

5. **Dependencies and Prerequisites**

Technical Requirements: Python, FastAPI, OpenAI API, Langchain, Docker

Infrastructure Requirements: Server, database, network connectivity

Personnel Requirements: Development team, testing, devops, business analyst and support team

6. **Success Criteria and Metrics**

Code Generation Quality: Accuracy, completeness and readability

Performance: Response time, throughput and scalability

User Adoption: Usage rates and user satisfaction

Maintenance Efficiency: Time-to-resolve issues and update frequency

7. **Future Enhancement**

LCNC whole application is a research analysis based iterative approach required to follow. First release we expected for a very small few page

application for non technical users who can't code to develop her own application. It will also help freelancers who can create the primary app and application back end before customising, optimising the functionalities and features.

Now back end code generated by this Code Engine needs to test properly, fix the issues or connections etc before devops team put this whole backend into a container with another database container to work perfectly as an isolated application only for the specific client's app.

There are the following points to optimise

Automate the whole process from generating the back end code to generate the database and put them into two containers in a separate place for hosting in a cloud server.

Auto testing after completing the back end code generation.

Auto deployment of back end code for a specific client app.

8. **Project Management**

Define project scope for each release: Identify application requirements, user needs and business objectives. Also, the feasibility of the requirement is very important in respect of the resources, team size and their expertise.

Assess risks: Identify potential technical, security and integration risks. Work closely with DevOps, Developer, Testing and analyst team.

Control the use of plugins and third party libraries: LCNC project overall including this Code Engine module using OpenAI and other AI libraries and modules extensively. So the project manager has the responsibility to regularly check the usage, cost and which team is using how much. Based on the expectation I need to provide the resources as well before they are required. So that dont waste the development time.

9. **Learning**

This is very important for the whole team and management to learn the usage of AI especially for Generative AI. How to use, feasible and capabilities of the team. Without the proper and right knowledge of the latest technologies including AI it is not possible to develop this kind of application.

Learning resources

https://python.langchain.com/docs/integrations/llms/openai/

https://python.langchain.com/docs/tutorials/rag/

https://www.youtube.com/watch?v=I_4jEnDwGwI

https://fastapi.tiangolo.com/deployment/docker/