FlowX: A Visual Building Tool that Merges Static Chatbot Flows with Dynamic LLM Responses

Izzat Alsharif
Amjad Kayed

Supervisor: Dr. Amjad Abu Hassan

January 31, 2025

Contents

List of Figures

Abstract

FlowX is a visual chatbot development platform that bridges the gap between static flow-based and dynamic LLM-powered conversation systems.

The platform features a drag-and-drop interface where developers can combine traditional decision trees with AI-driven interactions using specialized node types, including a smart switch capability for dynamic routing based on user intent. With FlowX, developers can create customizable chatbots, advanced automation systems, and more.

Built on a microservices architecture with ASP.NET Core, Python/LangChain, and React Native, FlowX enables the creation of hybrid chatbots that maintain both predictability and adaptability. FlowX demonstrates improved chatbot development efficiency in domains requiring both structured workflows and natural language understanding.

Chapter 1

Introduction

1.1 Background

Chatbots have become a cornerstone of modern digital interactions, bridging the gap between businesses and users through automated, yet personalized communication. Initially designed as rule-based systems with rigid decision trees, chatbots have evolved into dynamic agents powered by artificial intelligence (AI) and natural language processing (NLP). Despite this evolution, the current landscape of chatbot development tools remains fragmented. Solutions today are polarized: they either focus on static, flow-based designs (e.g. ManyChat, Chatfuel) or fully AI-driven interactions (e.g., Dialogflow, Rasa), with few attempts to integrate both approaches effectively. This divide limits developers' ability to create chatbots that are both predictable in structured workflows and adaptable to dynamic user input. Furthermore, existing tools often lack critical features such as advanced customization, extensibility through marketplaces, and flexible deployment options—gaps that hinder innovation and scalability in chatbot development.

1.2 Terminology

To ensure clarity throughout this report, key terms and concepts are defined below:

1.2.1 Core Concepts

- **Chatbot**: A software application designed to simulate human conversation through text or voice interactions, often used for customer support, information retrieval, or task automation.
- **Static Workflow**: A predefined conversation path using decision trees and fixed responses, offering predictable and structured interactions.
- **Dynamic Routing**: The ability to determine conversation flow based on AI analysis of user intent rather than predefined rules.
- **Hybrid Approach**: The combination of static workflows with dynamic, AI-powered interactions in a single chatbot system.

1.2.2 Technology Stack

- **LLM (Large Language Model)**: Advanced AI models trained on vast datasets to generate human-like text and understand context.
- **LangChain**: A framework for developing applications powered by language models, used in our executor service.
- **Microservices**: An architectural style where the application is structured as a collection of loosely coupled services.
- **gRPC**: A high-performance RPC (Remote Procedure Call) framework used for communication between services.
- **Infrastructure as Code (IaC)**: The practice of managing and provisioning infrastructure through code rather than manual processes.

1.2.3 Development Concepts

- **No-Code/Low-Code**: Development approach that enables users to create applications through visual interfaces rather than traditional programming.
- **Visual Builder**: A drag-and-drop interface for creating and modifying chatbot workflows without coding.
- **Visual Editor**: Interface for customizing the chatbot's appearance and user interface elements.

1.3 Objectives

The primary objectives of this work are:

- **To analyze existing chatbot development tools** and identify gaps in their ability to combine static workflows with dynamic adaptability, customization, extensibility and deployment options.
- **To design and develop FlowX**, a visual building tool that integrates static, rule-based workflows with dynamic, LLM-powered interactions, addressing the identified gaps.
- **To provide advanced customization tools** for chatbot design, allowing users to customize layouts, colors, fonts, and interactive elements to align with brand identity and user experience goals.
- **To create a marketplace ecosystem** for sharing, discovering and extending chatbot functionalities, fostering collaboration and reuse.
- **To simplify deployment** through flexible publishing options, including API integration and marketplace listings, ensuring accessibility across platforms.

1.4 Summary

FlowX addresses critical gaps in the chatbot development landscape by offering a unified platform that combines structured workflows with dynamic adaptability. Its emphasis on customization, extensibility, and ease of use makes it accessible to both technical and non-technical users, democratizing the creation of intelligent chatbots. By bridging the divide between static and AI-driven approaches, FlowX has the potential to improve scalability, reduce development costs, and improve user engagement across industries such as e-commerce, healthcare, and customer service.

Chapter 2

Literature Review

2.1 Introduction

Chatbot building tools have evolved from simple rule-based systems to AI-driven dynamic agents. However, most existing solutions focus either on static flow-based designs or fully AI-driven interactions, lacking a seamless integration of both approaches. This chapter reviews existing chatbot builders, automation platforms, and AI-driven chatbot frameworks to identify the gaps that FlowX aims to address.

2.2 Theoretical Foundations

2.2.1 Rule-based Chatbots

Rule-based chatbots follow predefined decision trees or workflows to respond to user input. These chatbots are deterministic and rely on explicit rules to generate responses. For example, a rule-based chatbot might ask a series of questions to gather information from the user and provide a specific response based on the answers. The next figure [1] shows an example of a rule-based chatbot conversation flow diagram. It consists of a series of nodes representing decision points and transitions between them based on user input.

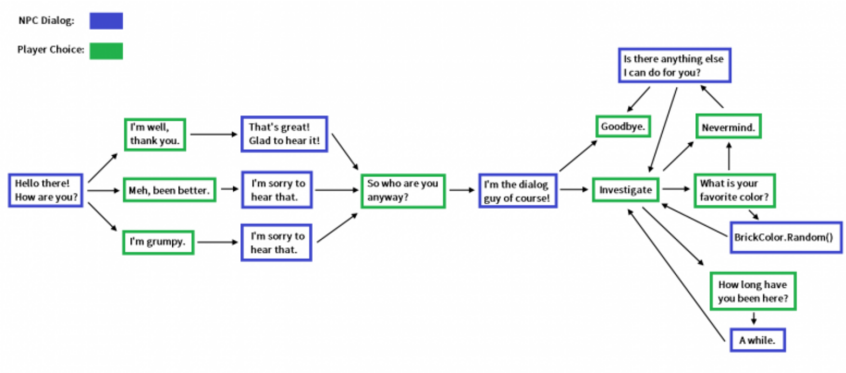

Figure 2.1: Example of a chatbot conversation flow diagram

2.2.2 Generative Chatbots

Generative chatbots leverage large language models (LLMs) like GPT-4 or Claude 3.5 Sonnet to produce contextually relevant responses. Unlike rule-based systems, these chatbots are probabilistic and generate outputs by learning patterns from training data, enabling flexible, open-ended interactions.

The next figure shows an example of a generative chatbot response generated by an LLM. In this example, the user input "explain how a combustion engine works" prompts the chatbot to generate a relevant response based on the context of the conversation.

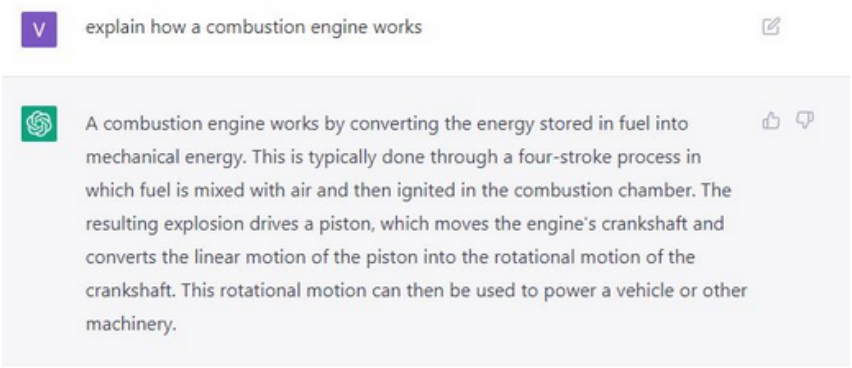

Figure 2.2: Screenshot of a message from ChatGPT.

2.2.3 Dynamic Routing

Dynamic routing refers to the process where a system, such as a chatbot powered by a large language model (LLM), dynamically selects one of several predefined options or pathways based on the context of the user input. Instead of following a rigid, rule-based decision tree, the LLM evaluates the input and chooses the most appropriate response or action from a set of predefined choices. This allows for more flexible and context-aware interactions while still maintaining some level of control over the chatbot's behavior.

The next figure shows an example of dynamic routing in a chatbot conversation. The user gives a textual input, and the LLM evaluates it to determine what option to choose out of two possible responses. This gives the flow structure and more control over the conversation while still allowing for dynamic user interactions.

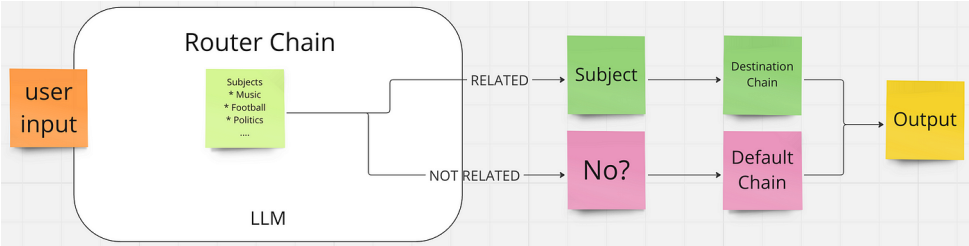

Figure 2.3: Example of dynamic routing in a chatbot conversation.

2.3 Existing Solutions and Their Approaches

2.3.1 No-Code Chatbot Builders

No-code chatbot builders, such as Chatfuel [2], Landbot [3], and ManyChat [4], allow users to create structured chatbot flows visually. These tools offer ease of use but are heavily reliant on predefined responses, which limits their adaptability to dynamic user input. For instance, they struggle to handle unexpected or unstructured queries, making them less suitable for complex conversational scenarios. While they excel in creating static, rule-based workflows, they lack the flexibility to adapt to more dynamic interactions. ManyChat, for example, is widely used for social media chatbots but is limited in its ability to handle complex, multi-turn conversations or integrate with advanced AI capabilities.

2.3.2 Automation and Workflow Platforms

Platforms like Zapier [5] and n8n [6] focus on workflow automation by connecting various services based on triggers and actions. While they excel in automating tasks, they do not offer built-in conversational capabilities or manage continuous user interactions. They can integrate with external tools, but they lack native support for multi-turn, context-aware conversations, limiting their use in chatbot development. These platforms are better suited for back-end automation rather than front-end conversational interfaces.

2.3.3 NLP-Powered AI Chatbots

Dialogflow [7] and Rasa [8] provide powerful NLP-driven chatbot solutions that utilize natural language processing (NLP) to understand user intent. These tools are highly adaptable and can handle dynamic user input, but often require significant technical expertise to set up and maintain. Furthermore, their reliance on AI and machine learning makes them unsuitable for users seeking simpler rule-based solutions. While they excel in handling unstructured conversations, they may be overkill and too advanced for simpler use cases.

2.3.4 Hybrid Models

Hybrid chatbot frameworks, such as Botpress [9] and Azure Bot Service [10], attempt to merge static workflows with dynamic adaptability. However, these solutions often require extensive manual setup and struggle to achieve a seamless balance between rule-based logic and dynamic interactions. For example, Botpress allows for custom scripting and integration with NLP models, but still prioritizes rule-based logic in many cases. Similarly, Azure Bot Service provides tools for integrating advanced capabilities, but the process can be complex and time-consuming, limiting its effectiveness for users who need simpler, more straightforward solutions.

2.4 Identified Gaps in Existing Solutions

The review of existing solutions reveals several key gaps in the current landscape of chatbot development tools:

- **Integration of Static and Dynamic Approaches**: While some tools excel in creating static, rule-based workflows (e.g., Chatfuel, Landbot) and others specialize in dynamic, AI-driven interactions (e.g., Dialogflow, Rasa), there is no unified solution that seamlessly combines the predictability of static chatbots with the adaptability of dynamic, LLM-powered responses. This fragmentation limits the ability to create chatbots that are both structured and flexible.

- **Limited Chatbot Customization**: Many platforms offer limited options for customizing the design and user interface of chatbots. For example, while tools like ManyChat allow for basic branding changes, they lack advanced customization features such as tailored layouts, interactive elements, or personalized themes. This restricts users from creating chatbots that align with their brand identity or user experience goals.

- **Customization and Extensibility**: Some platforms offer customization features, but they are often limited in scope or require technical expertise. Additionally, while certain tools allow for extensibility through integrations, there is no unified marketplace or ecosystem where users can easily share, discover, and extend chatbot functionalities in a collaborative manner.

- **Publishing and Deployment Options**: Many tools provide basic publishing options, such as embedding chatbots on websites, but lack flexible deployment methods like APIs or marketplace listings. This inconsistency limits the accessibility and reusability of chatbots across different platforms and use cases.

- **High Complexity in Setup and Maintenance**: Tools that offer advanced features, such as NLP or hybrid models (e.g., Botpress, Azure Bot Service), often require significant technical expertise and manual configuration. While these tools are powerful, their complexity creates a barrier for non-technical users who need simpler, more intuitive solutions.

2.5 Proposed Approach: FlowX

FlowX addresses these gaps by offering a unique approach to chatbot development that combines the strengths of static and dynamic workflows while providing additional features for customization, extensibility, and deployment. The key aspects of FlowX's approach include:

- **Seamless Integration of Static and Dynamic Capabilities**: FlowX enables users to create structured, rule-based chatbot flows while integrating dynamic, LLM-powered responses where needed. This hybrid approach allows for predictable, controlled interactions as well as adaptability to unstructured user input, bridging the gap between static and dynamic chatbot development.

- **Advanced Chatbot Customization**: FlowX provides extensive customization options for chatbot design, enabling users to tailor the look and feel of their chatbots to match their brand identity or user experience goals. Users can customize layouts, colors, fonts, and interactive elements, ensuring that chatbots are not only functional but also visually appealing and engaging.

- **Comprehensive Marketplace and Extensibility Features**: FlowX introduces a marketplace where users can discover, share, and extend chatbot functionalities. This ecosystem promotes collaboration and reusability, allowing users to customize their chatbots with pre-built components or share their own creations with the community. Unlike existing tools, FlowX provides a unified platform for both customization and extensibility.

- **Flexible Publishing Options**: FlowX provides multiple options for publishing chatbots, including deployment as APIs and listing on the marketplace. This flexibility ensures that chatbots can be easily integrated into various platforms and made accessible to a wider audience, addressing the limitations of existing tools that offer only basic publishing options.

- **User-Friendly Design**: FlowX prioritizes ease of use, enabling both technical and non-technical users to create, customize, and deploy chatbots without requiring extensive setup or maintenance. The platform's intuitive interface and guided workflows reduce the complexity often associated with chatbot development, making it accessible to a broader range of users.

By addressing these gaps, FlowX aims to provide a comprehensive solution that bridges the divide between static and dynamic chatbot development while fostering a collaborative ecosystem for innovation and reuse.

2.6 Conclusion

This literature review highlights the strengths and limitations of existing chatbot building tools, automation platforms, and AI-driven frameworks. Although some tools offer partial solutions, none provide a comprehensive approach that integrates structured flow-building with the power of LLM-based responses, alongside features such as advanced customization, a marketplace, robust extensibility, and flexible publishing options. FlowX addresses these challenges by combining the best of both worlds, creating a more flexible, intelligent, and user-friendly chatbot development experience.

Chapter 3

Methodology

3.1 Chatbot Builder

3.1.1 Initial Design

The Chatbot Builder is a web application that allows users to create chatbot workflows using a visual interface. This interface consists of a canvas where users can drag and drop nodes to create a flowchart-like structure. Each node represents a step in the chatbot conversation, and connections between nodes represent the flow of the conversation.

The following figure shows our initial design for the Chatbot Builder interface:

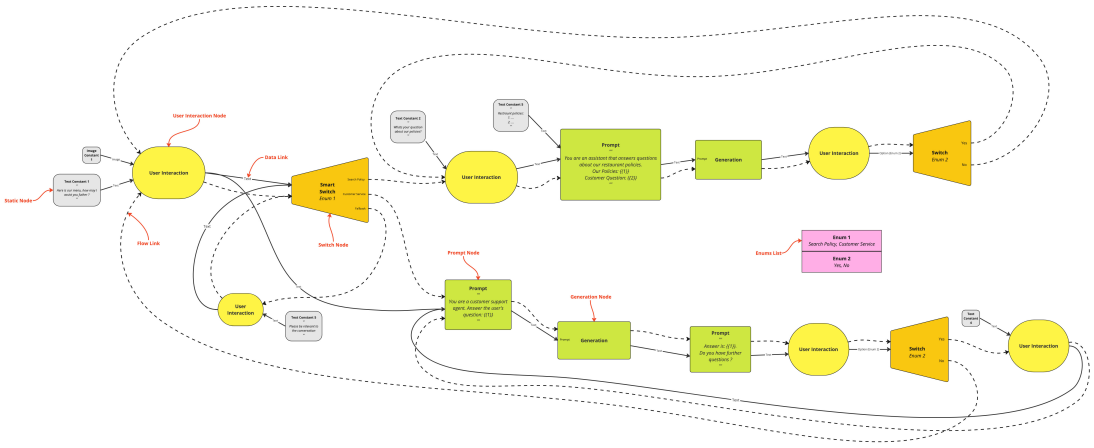

Figure 3.1: Initial design of the Chatbot Builder interface

The design shows the following components:

- **User Interaction Node**: Represents a user input node where the chatbot waits for user input.
- **Switch Node**: Represents a conditional node where the chatbot can branch based on a given option.
- **Prompt Node**: Represents a text template with placeholders for dynamic content at runtime.

- **Generation Node**: Represents a node that generates dynamic content using a language model.
- **Static Node**: Represents a constant value that is passed to other nodes.
- **Smart Switch Node**: Represents a conditional node that can branch based on the output of a language model, AKA Dynamic Routing.
- **Data Link**: Represents a connection between nodes that carries data from one node to another.
- **Flow Link**: Represents a connection between nodes that defines the flow of the conversation.

3.1.2 Design Decisions

- In this design we had to add two types of links, Data Link and Flow Link, this is because data is not always passed between nodes in a linear fashion, sometimes data is passed between nodes that are not directly connected in the flow of the conversation.
- To ensure logical easy to understand design, we choose an Eager-Evaluation approach where the data is passed through data links as soon as the node finishes its execution. Lazy-Evaluation approach is hard to understand and debug, as the data is passed only when needed, which makes it hard to track the data flow.
- Also notice that in this design we have exactly one User node (Interaction Node) instead of separate Input and Output nodes, this is because the chatbot follows a request-response model, where the user sends a request and the chatbot responds with a message, so there won't be any case where the user input isn't followed directly.

3.1.3 Conversation Flow

On Creation Request

- All static nodes are visited (In the order of their IDs), and their values are pushed through their output data links.
- Static nodes are of no use after that through out the whole conversation.
- Then the start (User Interaction) node is activated by returning output to user in the response, then the server waits for the next request.

On Further Requests

- Flow control resumes from the user interaction node which the server stopped on last time.
- User input is pushed through the node's output data links.
- Flow control moves to the next node through the node's flow link.

- The following nodes consecutively: activate, push their data, and move flow control. Until the server eventually hit another user interaction node.
- Output is finally returned to the user in the response. And the server waits for the next request.

3.2 System Design and Architecture

3.2.1 Overview of the System

The following figure shows an overview of the system architecture of FlowX. Refer to each microservice's section for more details on the implementation.

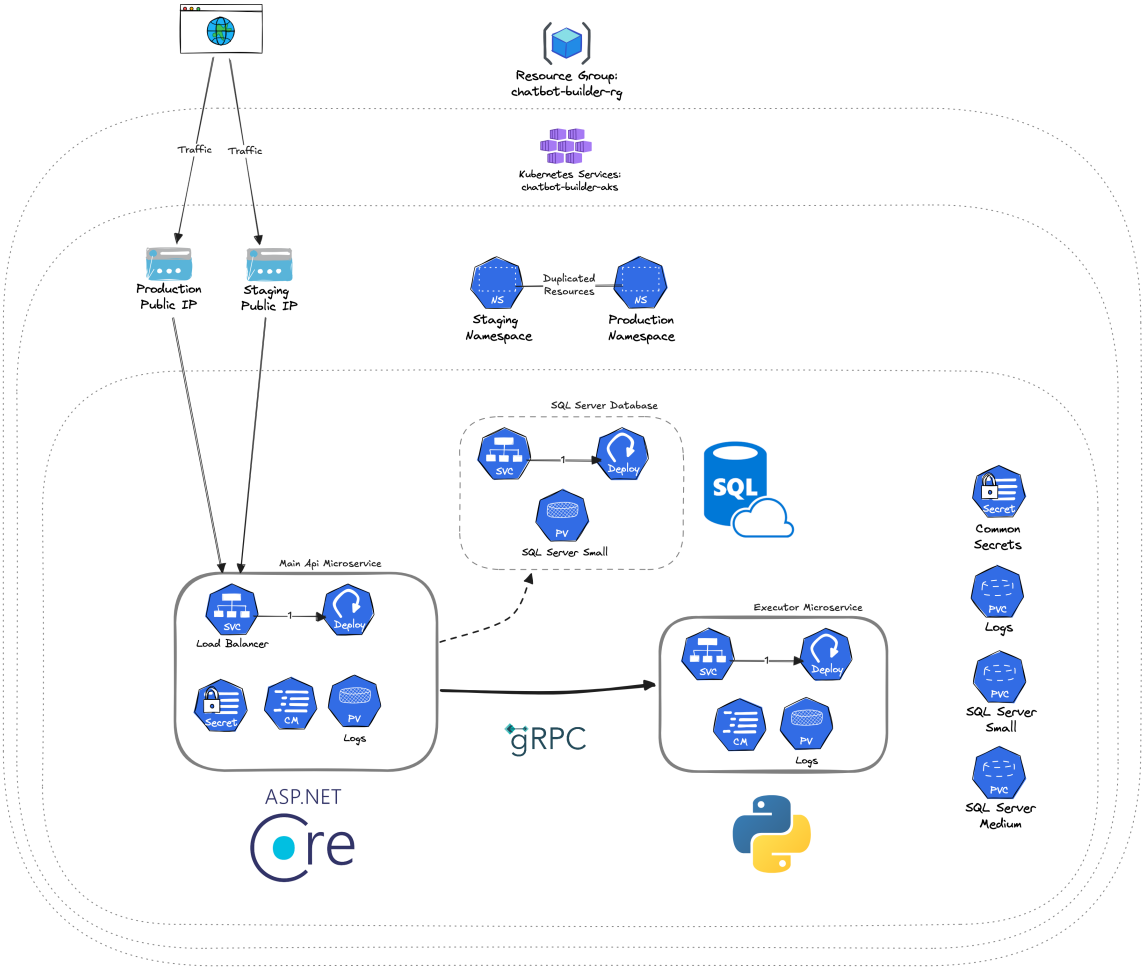

Figure 3.2: Microservices architecture of FlowX

The project was developed using a microservices architecture, with separate repositories for each microservice and the frontend client. The repositories are as follows:

- **chatbot-builder-api**: The main API microservice for the Chatbot Builder application. Responsible for authentication and orchestrating workflows traversal & state-management, uses the Executor-Service for LangChain logic.

- **chatbot-builder-executor**: The LangChain execution microservice for the Chatbot Builder application. Controlled by the API-Service, responsible for executing LangChain logic in workflows.
- **chatbot-builder-infra**: Infrastructure & deployment of the Chatbot Builder application using Terraform & Kubernetes.
- **chatbot-builder-client**: The frontend for the Chatbot Builder application, built with React Native.
- **chatbot-builder-protos**: Protocol Buffers schema definitions for gRPC communication between API and Executor services.

3.2.2 API Microservice

This is the main API microservice for the Chatbot Builder application. It is responsible for authentication and orchestrating workflows traversal & state-management.

Implemented using ASP.NET Core, which provides a robust strongly typed framework that is easy to maintain and scale.

The following is a few of the key features of the API microservice:

- Authentication: JWT-based authentication for user management.
- Workflow Management: CRUD operations for workflows, nodes, and connections.

3.2.3 Executor Microservice

This is the LangChain execution microservice for the Chatbot Builder application. It is controlled by the API-Service and is responsible for executing LangChain logic in workflows.

Implemented using Python, which contains the Langchain library that abstracts LLM model management and execution.

The following is a few of the key features of the Executor microservice:

- gRPC Service: Allows for calls from the API service to execute LangChain logic.

3.3 Backend

3.3.1 Overview

The backend of FlowX uses a microservices architecture for scalability and modularity. It consists of:

- **API Microservice**: Handles authentication, workflow, and state management.
- **Executor Microservice**: Executes LangChain logic.
- **Infrastructure Service**: Manages deployment with Terraform and Kubernetes.

The following figure shows the API schema for the backend that allows Workflow creation:

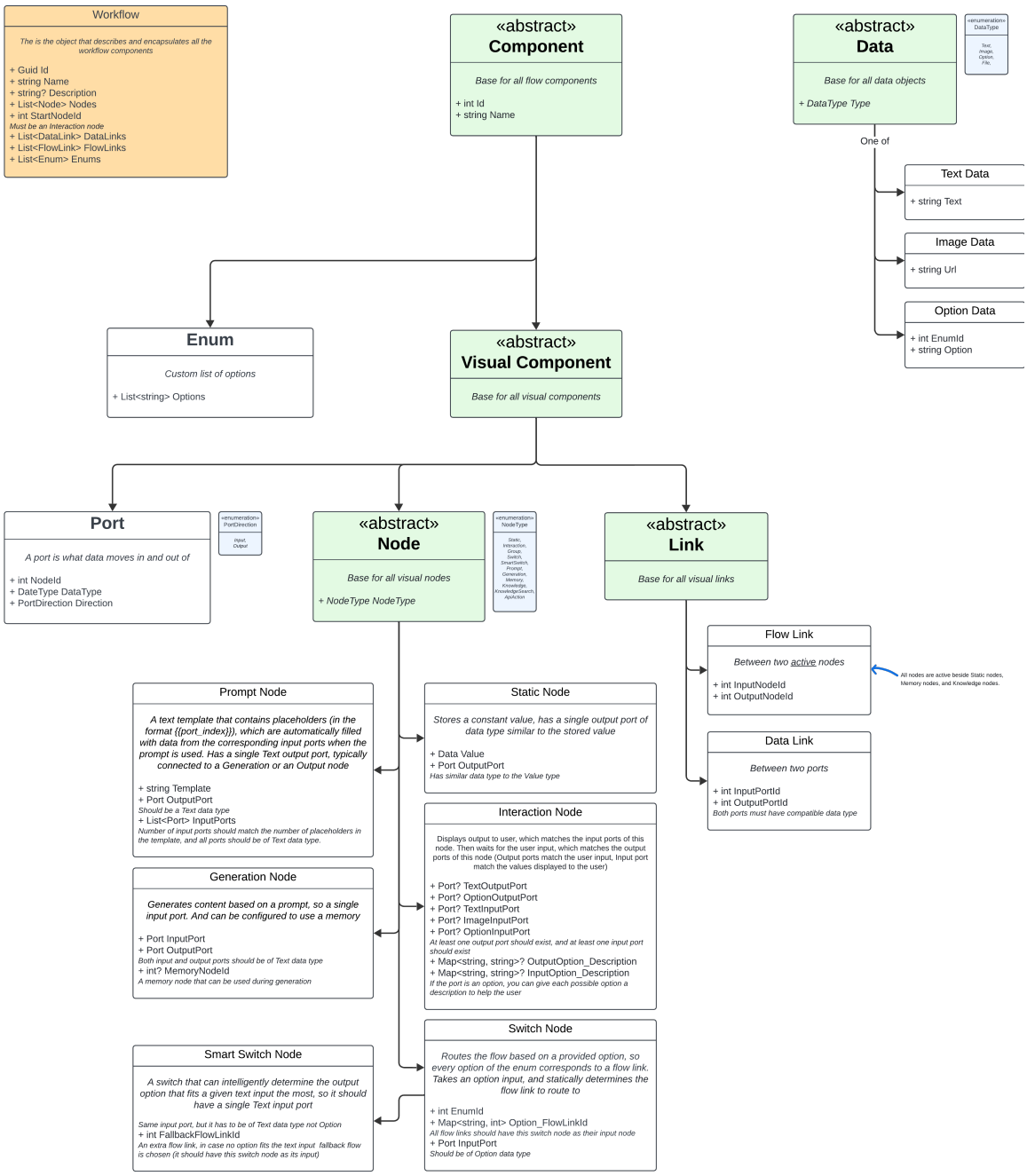

Figure 3.3: API Schema for Chatbot Builder Workflows

3.3.2 API Microservice

Built with **ASP.NET Core**, this service handles:

- **Authentication**: Uses JWT-based security.
- **Workflow Management**: Supports CRUD operations.
- **State Management**: Tracks chatbot workflow states.
- **Executor Communication**: Delegates execution via gRPC.

3.3.3 Domain Model

The API service implements a rich domain model to support both user management and chatbot workflow functionality. Figure 3.4 shows the core domain entities and their relationships.

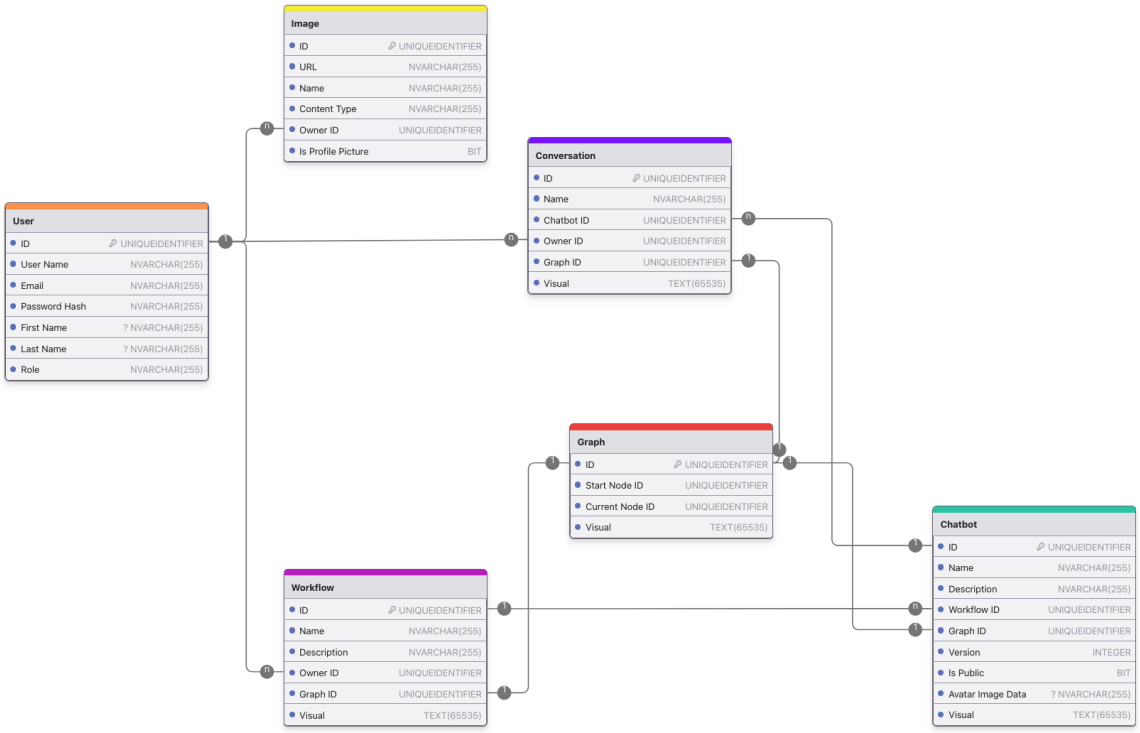

Figure 3.4: Core Domain Model Class Diagram

The domain model is centered around these key entities:

- **User**: Manages authentication and profile data, including role-based access control
- **Chatbot**: Encapsulates chatbot configuration and references to its workflow implementation
- **Workflow**: Contains the complete chatbot logic as a graph structure
- **Graph**: Maintains the topology of nodes and connections that define the chatbot's behavior
- **Conversation**: Handles state management for ongoing user interactions

3.3.4 Graph Implementation

The workflow execution engine is built on a sophisticated graph structure (Figure 3.5) that enables both static flows and dynamic LLM-powered interactions.

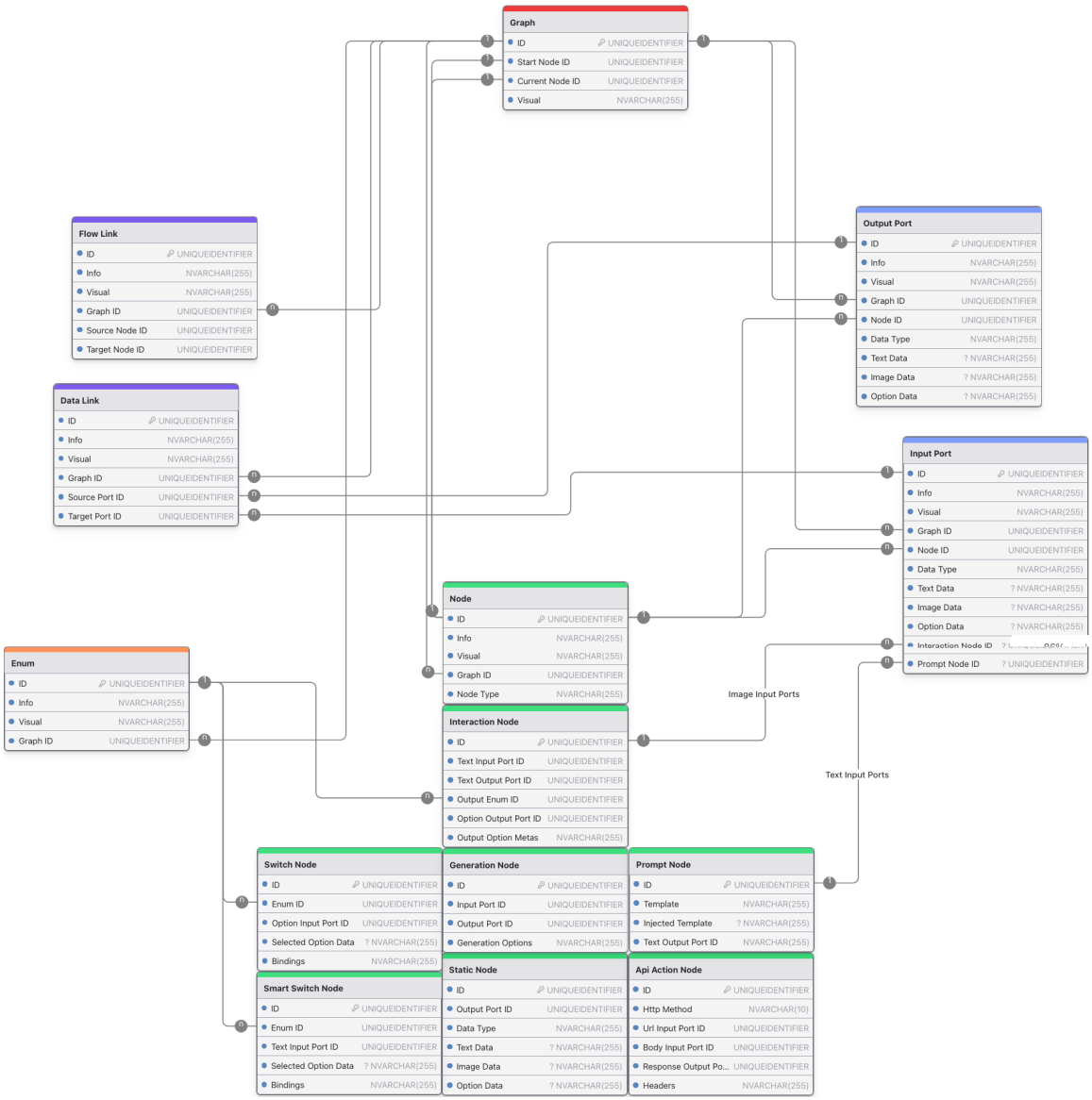

Figure 3.5: Graph Implementation Class Diagram

The graph implementation includes:

- **Node**: Abstract base class defining common node behavior
- **Input/Output Ports**: Strongly-typed connection points enabling type-safe data flow
- **Flow/Data Links**: Distinct connection types for control flow vs data transfer
- **Specialized Nodes**:
  - **Interaction Node**: Manages user input/output with templating support
  - **Generation Node**: Interfaces with the Executor service for LLM operations
  - **Switch Node**: Implements both static and LLM-powered conditional branching
  - **Static Node**: Provides deterministic responses
  - **API Action Node**: Enables integration with external services

3.3.5 Executor Microservice

Developed in **Python** with **LangChain**, responsible for:

- **Chatbot Logic Execution**.
- **gRPC Communication** with the API service.
- **Handling Smart Switch Nodes** using LLM outputs.

3.3.6 Infrastructure and Deployment

Managed using **Terraform** and **Kubernetes**, ensuring:

- **Scalable Deployment**: Kubernetes handles scaling and failover.
- **Infrastructure as Code (IaC)**: Terraform automates provisioning.
- **Service Communication**: Uses gRPC and RESTful APIs.

3.4 Frontend

3.4.1 Overview and Architecture

The frontend of FlowX is built with **React Native**, providing a cross-platform mobile development solution. The application follows a component-based architecture with clear separation of concerns:

- **Presentation Layer**: React Native components and screens
- **State Management**: Redux store with middleware for side effects
- **Service Layer**: API client and WebSocket handlers
- **Utils**: Helper functions and shared utilities

3.4.2 Visual Workflow Builder

The core of the application is the visual workflow builder, implemented with several specialized components:

- **Canvas Management**:
  - Infinite canvas with pan and zoom capabilities
  - Grid-based snapping for precise node placement
  - Multi-select and bulk operations
- **Node System**:
  - Drag-and-drop node creation and positioning
  - Smart port connection system with type validation
  - Real-time node state visualization

  - Custom node templates for different interaction types
  - Bezier curve connections with interactive editing

- **Node Editor**:
  - Rich text editor for message templates
  - Dynamic form generation for node configuration
  - Each node with its own properties

- **Chat Designer**:
  - Live preview of chatbot behavior
  - Mobile-first chat interface
  - Allows for designing the visuals of the chatbot

3.4.3 State Management and Performance

The application uses Redux for state management with several optimizations for performance:

- **Store and Updates**:
  - Normalized state shape for efficient updates
  - Separate slices for UI state and domain data
  - Optimistic updates for better responsiveness
  - Redux Thunk for async operations
  - Persistence middleware for auto-saving

- **Performance Optimizations**:
  - Virtual scrolling for large workflows
  - Lazy loading of node components
  - Memoization of expensive computations
  - Batched updates for multiple changes
  - Debounced save operations
  - Selective re-rendering using React.memo

3.4.4 Cross-Platform Design

The application ensures consistent behavior across platforms through:

- **Responsive Design**:
  - Platform-specific UI adjustments
  - Adaptive layouts for different screen sizes

3.5 Infrastructure

3.5.1 Architectural Approach

The infrastructure was designed using Infrastructure-as-Code (IaC) principles with three core components:

- **Terraform**: For provisioning cloud resources on Azure, leveraging Terraform Cloud for state management.
- **Kubernetes**: For container orchestration and deployment management using Azure Kubernetes Service (AKS).
- **GitHub Actions**: For CI/CD pipeline automation, managing infrastructure provisioning and application deployment.

This combination enables reproducible environments and automated deployment processes across staging and production environments.

3.5.2 Terraform Implementation

The Terraform configuration manages the complete Azure infrastructure through modular components:

- **Resource Groups**: Logical containers for Azure resources.
- **AKS Clusters**: Kubernetes clusters with auto-scaling node pools.
- **Blob Storage**: Azure Blob Storage for persistent data storage.
- **Public IPs**: Dedicated IP addresses per environment, injected into Kubernetes manifests.

```
module "aks" {
  source              = "./modules/aks"
  environment         = "staging"
  node_count          = 3
  vm_size             = "Standard_D2_v2"
  dns_prefix          = "chatbot-builder-staging"
  resource_group_name = azurerm_resource_group.main.name
}
```

Listing 3.1: Sample Terraform module for AKS

Terraform outputs include:

- **Kubernetes Configuration**: Kubeconfig file for cluster authentication.
- **Public IPs**: Used for Load Balancers and injected into Kubernetes manifests.

3.5.3 Kubernetes Orchestration

The Kubernetes implementation follows a multi-environment strategy using Kustomize:

- **Base Configuration**: Common resources (Pods, Services, Ingress).
- **Environment Overlays**: Staging and Production-specific customizations.
- **Secret Management**: Kubernetes secrets encrypted with SealedSecrets.

Deployment follows this phased approach:

1. Persistent Volume Claims for storage.
2. ConfigMaps for environment variables.
3. Microservice deployments.
4. Network policies and ingress rules.

Directory structure:

- **base**/: Core manifests including ConfigMaps, PVCs, Deployments, and Services.
- **overlays**/: Staging and production configurations.
- **secrets**/: Templates for Kubernetes secrets, applied manually before automation.

3.5.4 CI/CD Pipeline

The CI/CD pipeline is managed with GitHub Actions and follows GitOps principles. Key workflows:

- **Terraform Deployment**: Runs on changes to the ‘infra/‘ directory.
  - Initializes Terraform and validates configuration.
  - Applies infrastructure changes if on ‘main‘ branch.
  - Outputs kubeconfig and public IPs.
- **Kubernetes Deployment**: Triggered via repository dispatch for staging/production.
  - Fetches Terraform outputs (Kubernetes config, public IPs).
  - Updates Kubernetes manifests with correct environment values.
  - Deploys microservices and infrastructure resources to AKS.

3.6 Project Management

The project was managed using Jira [11], a widely adopted Agile project management tool. The development process followed an iterative approach, with tasks organized into epics that represent the major functional areas. Key epics included Technical Setup, Prototype Development, Feature Implementation, and Documentation.

The project was divided into several epics, each representing a major functional area or phase of development. Below is a detailed breakdown of the epics and their associated tasks:

Epic 1: Technical Aspects

This epic focused on setting up the foundational technical infrastructure for the project.

- CB-4: Setup CI/CD with AKS
- CB-6: Setup API-Service Project
- CB-5: Setup Executor-Service Project
- CB-18: Setup Engine-Service Project
- CB-20: Add SqlServer Database to API Service
- CB-21: Setup Terraform Project
- CB-17: Maintain Kubernetes Manifests
- CB-22: Maintain CI/CD Pipeline
- CB-23: Maintain Terraform Infrastructure
- CB-38: Add MongoDB to Executor Service

Epic 2: Prototype Development

This epic involved creating the initial prototype and implementing core functionalities.

- CB-8: Creating the First Prototype
- CB-9: Setup Jwt Authentication in the Api-Service
- CB-10: Develop Api Schema
- CB-11: Develop gRPC Schema between Api-Service & Engine-Service
- CB-19: Implement Engine-Service Domain
- CB-24: Implement Engine-Service Application
- CB-13: Implement Api-Service Logic
- CB-25: Merge Engine into API Temporarily

Epic 3: Adding Core Features

This epic focused on adding essential features to the platform.

- CB-26: Adding Essential Features
- CB-27: Implement User Images Endpoints
- CB-29: Add More User Fields
- CB-28: Maintain Api Endpoints
- CB-30: Add ApiAction Node

- CB-12: Develop gRPC Schema between Engine-Service & Executor-Service
- CB-32: Add Generation & SmartSwitch Nodes
- CB-14: Implement Executor-Service Logic
- CB-33: Add Image Ports for Interaction Node

Epic 4: Adding Advanced Features

This epic involved implementing additional advanced features to enhance the platform's capabilities.

- CB-35: Adding Extra Features
- CB-37: Add ImageData to InteractionOptionMetas
- CB-36: Add Avatar Picture to Chatbots
- CB-41: Return ProfilePicture with User details
- CB-42: Add Object Visual fields
- CB-43: Add Chatbot-Graph Query for Owners
- CB-31: Add Entities Statistics

Epic 5: Domain Aspects

This epic focused on project documentation, reporting, and final deliverables.

- CB-7: Maintain Pages & ReadMes
- CB-15: Add Workflow Examples
- CB-16: Write Project Report
- CB-39: Create Demo & Presentation

Chapter 4

Results and Discussion

4.1 System Implementation

The implemented FlowX platform successfully delivers a comprehensive chatbot development environment. This section presents the key interfaces and functionality of the system, followed by an analysis of its performance and effectiveness. The complete source code for all components is available in the project's GitHub organization[1], which contains the following repositories:

- **chatbot-builder-api**: API Gateway for authentication and service orchestration
- **chatbot-builder-client**: Frontend built with React Native
- **chatbot-builder-executor**: LangChain execution service
- **chatbot-builder-infra**: Infrastructure code using Terraform and Kubernetes
- **chatbot-builder-protos**: Protocol Buffer definitions for gRPC communication

4.2 User Interface and Development Flow

4.2.1 Homepage and Dashboard

The FlowX platform provides an intuitive entry point for users through its homepage interface (Figure 4.1).

[1] `https://github.com/Chatbot-Builder-Project`

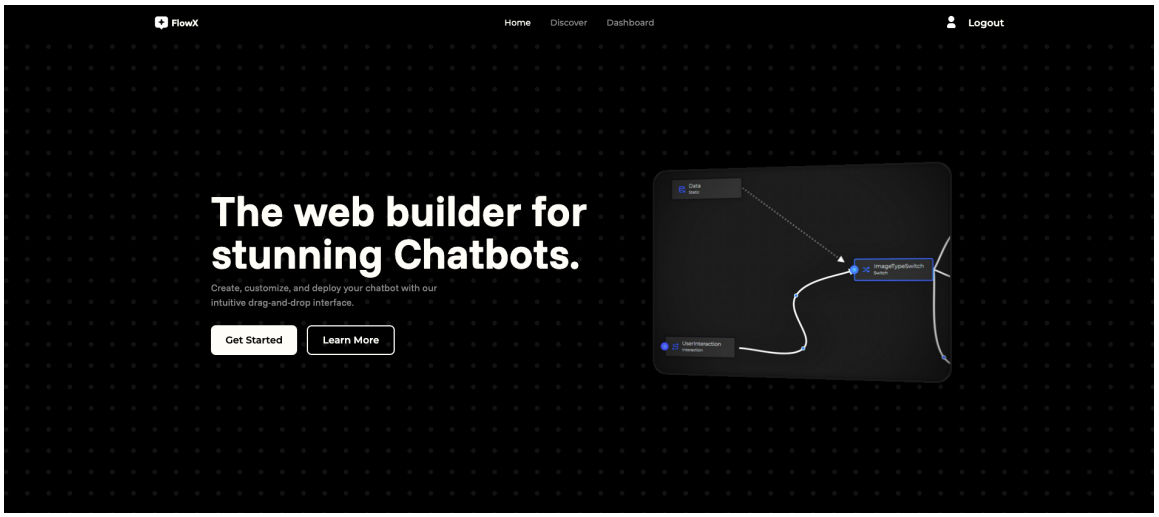

Figure 4.1: FlowX Platform Homepage

4.2.2 Workflow Creation

Users begin by creating a new chatbot project through a streamlined workflow creation process (Figure 4.2).

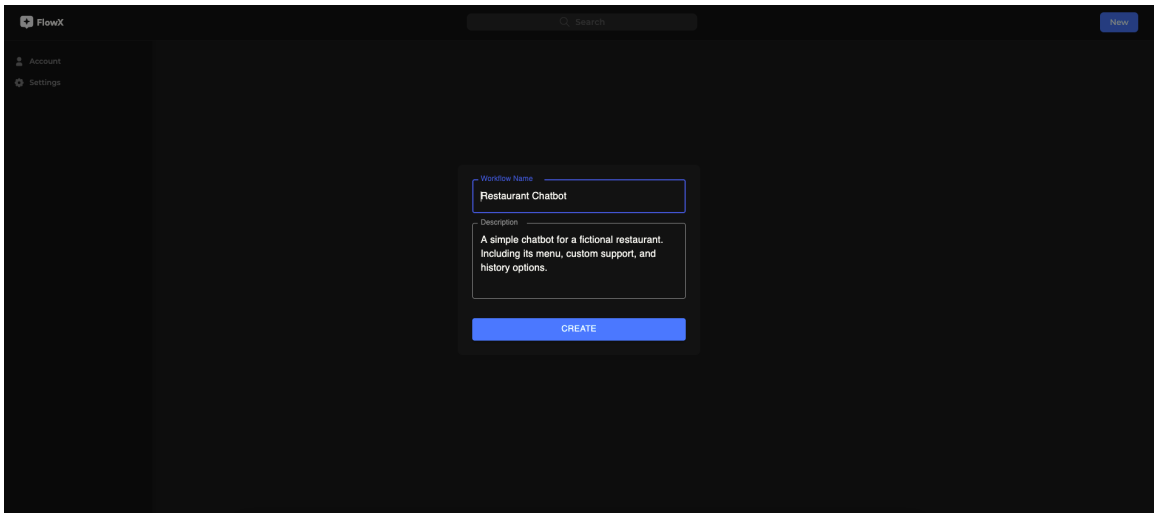

Figure 4.2: Initial Workflow Creation Interface

4.2.3 Workflow Builder

After creation, users can design their chatbot's conversation logic using the workflow builder. The system provides a simple starter workflow (Figure 4.3) that users can build upon. Through the workflow builder's capabilities, users can develop sophisticated conversation patterns, as demonstrated by the complex implementation in Figure 4.4.

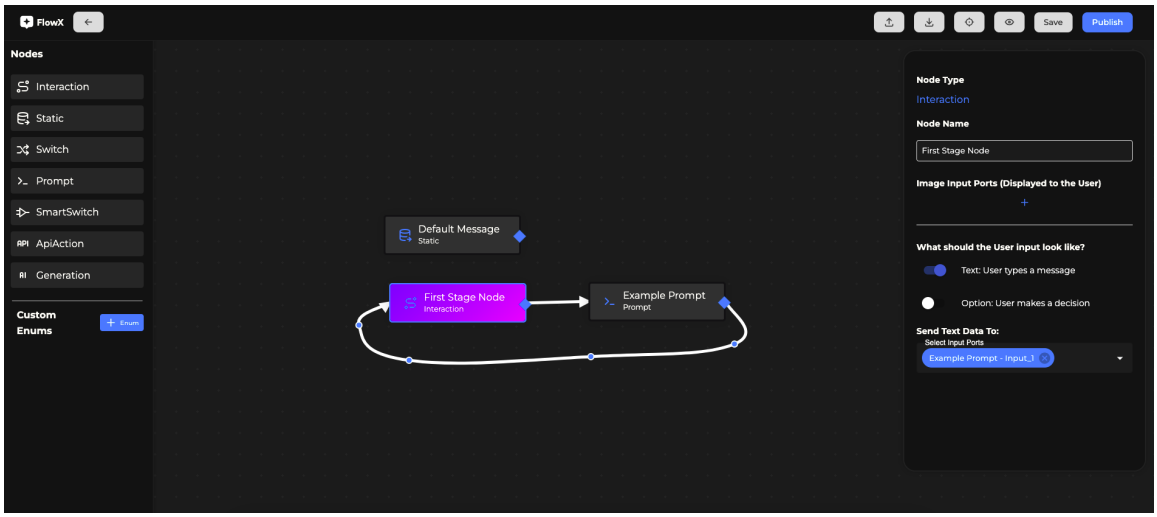

Figure 4.3: Initial Default Workflow State

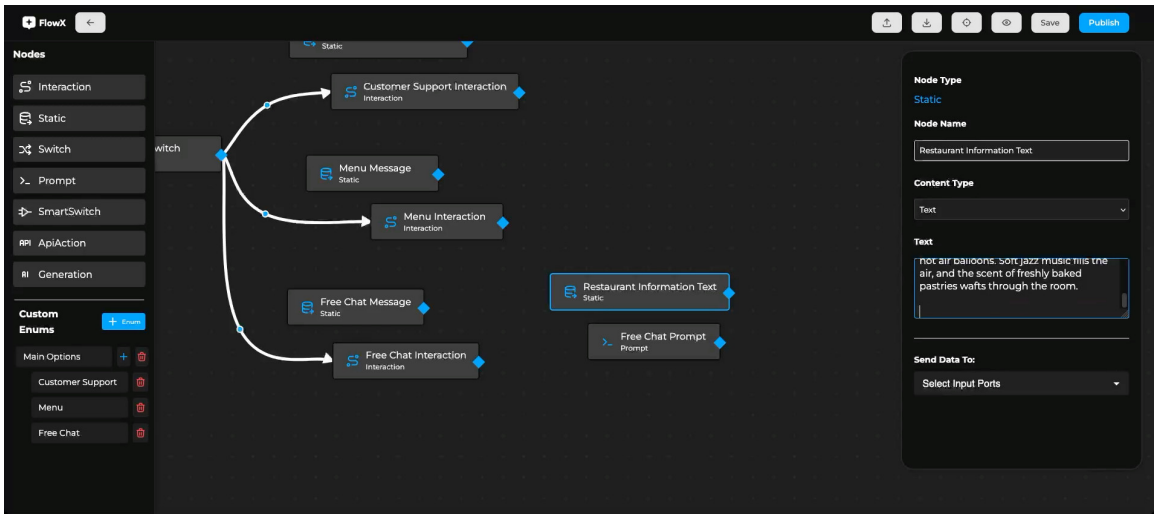

Figure 4.4: Example of an Advanced Workflow Built Using the Platform

4.2.4 Visual Editor

Once the conversation logic is defined, developers can customize the chatbot's visual appearance using the visual editor (Figure 4.5). This interface allows for detailed customization of how the chatbot will appear to end users.

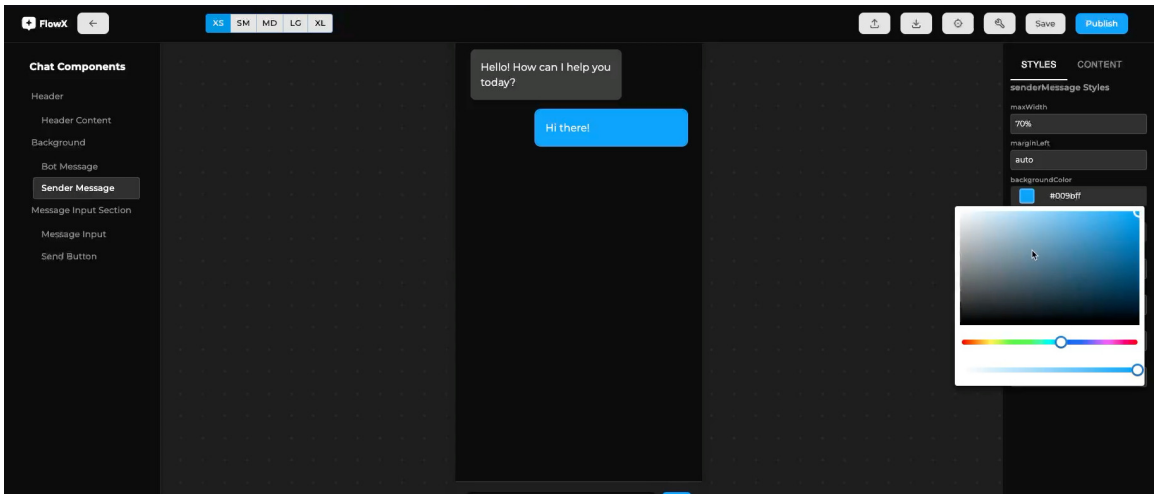

Figure 4.5: Chatbot Visual Customization Interface

4.2.5 Live Testing Interface

The platform includes a conversation testing interface (Figure 4.6) that allows developers to validate both their chatbot's logic and visual appearance in real-time.

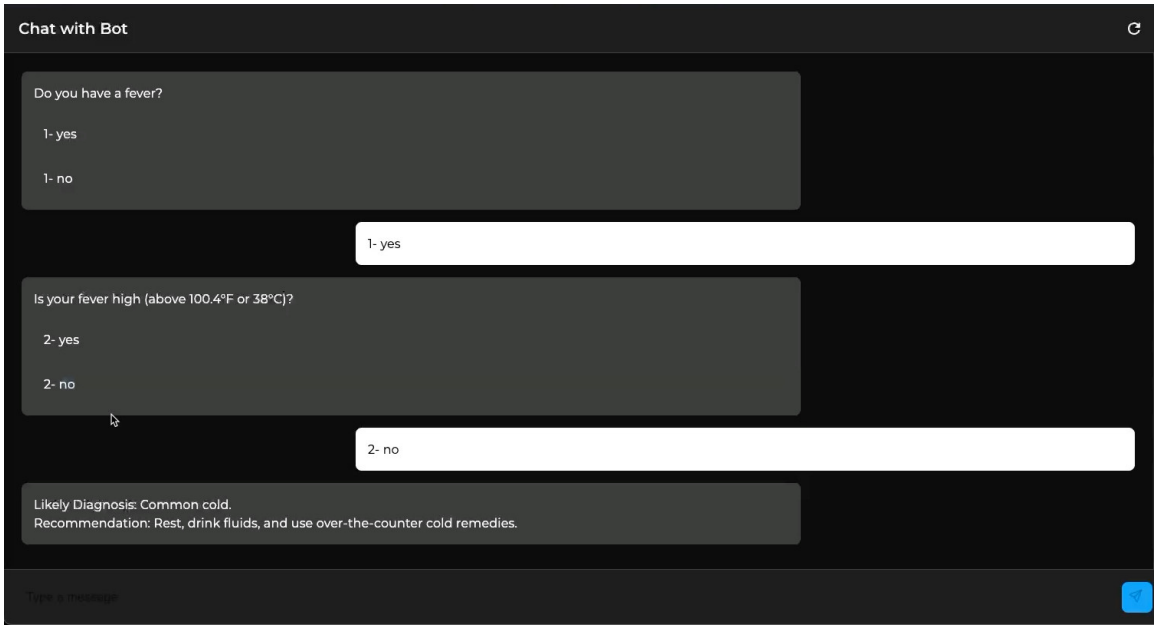

Figure 4.6: Live Conversation Testing Interface

Appendix A

Source Code Repositories

The complete source code for the FlowX platform is organized across multiple repositories within the Chatbot Builder Project organization on GitHub (`https://github.com/Chatbot-Builder-Project`). This appendix provides details about each repository's purpose and contents.

A.1 Repository Structure

- **chatbot-builder-api**
  - Main API Gateway service
  - Handles authentication and authorization
  - Manages workflow execution and state
  - Implements the domain model and business logic
  - Built with ASP.NET Core
- **chatbot-builder-client**
  - Frontend application
  - Implements the workflow builder interface
  - Provides visual customization tools
  - Built with React Native for cross-platform support
- **chatbot-builder-executor**
  - LangChain execution service
  - Handles LLM integration and dynamic routing
  - Implements the Python-based execution engine
  - Communicates with API via gRPC
- **chatbot-builder-infra**
  - Infrastructure as Code repository
  - Contains Terraform configurations for Azure resources

  - Includes Kubernetes manifests for deployment
  - Manages CI/CD pipelines with GitHub Actions

- **chatbot-builder-protos**
  - Protocol Buffer definitions
  - Defines service contracts for gRPC communication
  - Shared between API and Executor services

A.2 Development Workflow

The repositories follow a microservices architecture pattern, with each service independently versioned and deployed. The development workflow includes:

- Feature branches for development
- Pull request reviews for code quality
- Automated testing in CI/CD pipelines
- Automated deployment to staging and production

For detailed setup and contribution guidelines, refer to each repository's README file.

References

[1] Medium, "Chatbot conversation flow example," 2025, accessed: 2025-02-01. [Online]. Available: https://miro.medium.com/max/3116/1*Y5DokuM8QuKXaJdz12WtOw.png

[2] Chatfuel, "Chatfuel - no code chatbot builder," 2024, accessed: 2025-02-01. [Online]. Available: https://chatfuel.com/

[3] Landbot, "Landbot - conversational ai and chatbots," 2024, accessed: 2025-02-01. [Online]. Available: https://landbot.io/

[4] ManyChat, "Manychat - messenger marketing and chatbot platform," 2024, accessed: 2025-02-01. [Online]. Available: https://manychat.com/

[5] Zapier, "Zapier - automate your workflows," 2024, accessed: 2025-02-01. [Online]. Available: https://zapier.com/

[6] n8n, "n8n - open source workflow automation," 2024, accessed: 2025-02-01. [Online]. Available: https://n8n.io/

[7] G. Cloud, "Dialogflow - ai chatbot development," 2024, accessed: 2025-02-01. [Online]. Available: https://cloud.google.com/dialogflow

[8] Rasa, "Rasa - open source conversational ai," 2024, accessed: 2025-02-01. [Online]. Available: https://rasa.com/

[9] Botpress, "Botpress - open-source conversational ai platform," 2024, accessed: 2025-02-01. [Online]. Available: https://botpress.com/

[10] M. Azure, "Azure bot service - ai chatbots for businesses," 2024, accessed: 2025-02-01. [Online]. Available: https://azure.microsoft.com/en-us/products/bot-services/

[11] Atlassian, "Jira - issue & project tracking software," 2024, accessed: 2025-02-01. [Online]. Available: https://www.atlassian.com/software/jira/