**CONTENTS**

[CHAPTER 1: INTRODUCTION 6](#_Toc213869570)

[1.1 MOTIVATION 6](#_Toc213869571)

[1.2 PROBLEM STATEMENT 7](#_Toc213869572)

[1.3 OBJECTIVE 7](#_Toc213869573)

[1.4 SCOPE AND LIMITATIONS 8](#_Toc213869574)

[*1.4.1 In-Scope Features* 8](#_Toc213869575)

[*1.4.2 Out of Scope Features* 9](#_Toc213869576)

[1.5 APPLICATION AND TARGET AUDIENCE 9](#_Toc213869577)

[1.6 REPORT ORGANIZATION 10](#_Toc213869578)

[CHAPTER 2: LITERATURE SURVEY AND BACKGROUND STUDY 11](#_Toc213869579)

[2.1 THE PARADIGM SHIFT: THE RISE OF VISUAL WORKFLOW AUTOMATION 11](#_Toc213869580)

[2.2 THE "ABSTRACTION GAP" IN MACHINE LEARNING PRACTICE 11](#_Toc213869581)

[2.3 A CRITICAL REVIEW OF EXISTING VISUAL ML PLATFORMS 12](#_Toc213869582)

[*2.3.1 Desktop-Based Analytics Workbenches (KNIME, Orange)* 12](#_Toc213869583)

[*2.3.2 Enterprise-Grade Cloud Platforms (Azure ML, Vertex AI)* 13](#_Toc213869584)

[*2.3.3 Positioning DominoML* 13](#_Toc213869585)

[2.4 JUSTIFICATION OF FOUNDATIONAL TECHNOLOGIES 13](#_Toc213869586)

[**CHAPTER 3: SYSTEM REQUIREMENTS SPECIFICATION** 15](#_Toc213869587)

[3.1 FUNCTIONAL REQUIREMENTS 15](#_Toc213869588)

[3.2 NON-FUNCTIONAL REQUIREMENTS 17](#_Toc213869589)

[3.3 HARDWARE REQUIREMENTS (FOR HOSTING) 18](#_Toc213869590)

[3.4 SOFTWARE REQUIREMENTS 18](#_Toc213869591)

[*3.4.1 Development/Deployment Environment* 18](#_Toc213869592)

[*3.4.2 Client-Side* 18](#_Toc213869593)

[3.5 USER CHARACTERISTICS 18](#_Toc213869594)

[**CHAPTER 4: SYSTEM DESIGN AND ARCHITECTURE** 19](#_Toc213869595)

[4.1 HIGH-LEVEL SYSTEM ARCHITECTURE 19](#_Toc213869596)

[4.2 TECHNOLOGY STACK 21](#_Toc213869597)

[4.3 DATABASE DESIGN 22](#_Toc213869598)

[*4.3.1 Entity-Relationship (ER) Diagram* 22](#_Toc213869599)

[*4.3.2 Schema Definition* 23](#_Toc213869600)

[4.4 DESIGN FRAMEWORKS 24](#_Toc213869601)

[*4.4.1 Use Case Diagram* 24](#_Toc213869602)

[*4.4.2 Data Flow Diagrams (DFD)* 25](#_Toc213869603)

[4.5 FRONTEND DESIGN AND UI/UX 27](#_Toc213869604)

[*4.5.1 The Builder Interface Layout* 27](#_Toc213869605)

[*4.5.2 Design System and Color Palette* 28](#_Toc213869606)

[**CHAPTER 5: IMPLEMENTATION DETAILS** 30](#_Toc213869607)

[5.1 PROJECT STRUCTURE 30](#_Toc213869608)

[5.2 BACKEND IMPLEMENTATION (FLASK) 31](#_Toc213869609)

[*5.2.1 Application Factory and Configuration* 31](#_Toc213869610)

[*5.2.2 Database Models* 31](#_Toc213869611)

[*5.2.3 Authentication Module* 32](#_Toc213869612)

[*5.2.4 API Endpoints* 32](#_Toc213869613)

[5.3 FRONTEND IMPLEMENTATION (VANILLA JAVASCRIPT) 33](#_Toc213869614)

[*5.3.1 The Canvas Logic* 33](#_Toc213869615)

[*5.3.2 Component and Properties Management* 33](#_Toc213869616)

[*5.3.3 Undo/Redo System* 34](#_Toc213869617)

[*5.3.4 Version Management UI* 34](#_Toc213869618)

[5.4 CORE FEATURE IMPLEMENTATION 34](#_Toc213869619)

[*5.4.1 Code Generation Logic* 34](#_Toc213869620)

[*5.4.2 Real-time Validation* 35](#_Toc213869621)

[*5.4.3 Pipeline Versioning and Metric Tracking* 36](#_Toc213869622)

[**CHAPTER 6: SYSTEM TESTING** 37](#_Toc213869623)

[6.1 TESTING OBJECTIVES 37](#_Toc213869624)

[6.2 TESTING METHODOLOGIES 37](#_Toc213869625)

[*6.2.1 Unit Testing (Backend)* 38](#_Toc213869626)

[*6.2.2 Integration Testing (Backend)* 38](#_Toc213869627)

[*6.2.3 Manual End-to-End System Testing* 39](#_Toc213869628)

[*6.2.4 User Acceptance Testing (UAT)* 40](#_Toc213869629)

[6.3 SAMPLE TEST CASES 40](#_Toc213869630)

[6.4 TESTING ENVIRONMENT 42](#_Toc213869631)

[**CHAPTER 7: RESULTS AND SNAPSHOTS** 44](#_Toc213869632)

[7.1 APPLICATION SCREENSHOTS 44](#_Toc213869633)

[7.2 SAMPLE GENERATED CODE 45](#_Toc213869634)

[**CHAPTER 8: CONCLUSION AND FUTURE SCOPE** 50](#_Toc213869635)

[8.1 CONCLUSION 50](#_Toc213869636)

[8.2 LIMITATIONS OF THE CURRENT SYSTEM 51](#_Toc213869637)

[8.3 FUTURE SCOPE 51](#_Toc213869638)

[*8.3.1 Immediate Future Scope: Phase 3 - Export Runnable Artifacts* 51](#_Toc213869639)

[*8.3.2 Long-Term Enhancements* 52](#_Toc213869640)

[**REFERENCES** 53](#_Toc213869641)

[**APPENDIX A: CORE PYTHON CODE SNIPPETS** 54](#_Toc213869642)

[**APPENDIX B: CORE JAVASCRIPT CODE SNIPPETS** 58](#_Toc213869643)

[**APPENDIX C: ENVIRONMENT SETUP AND INSTALLATION GUIDE** 62](#_Toc213869644)

## CHAPTER 1: INTRODUCTION

Machine Learning (ML) has transitioned from a niche academic discipline to a cornerstone of modern technology, driving innovation in fields ranging from healthcare and finance to entertainment and logistics. The process of developing an effective ML model, however, is inherently complex and iterative. It involves a multi-stage workflow that includes data collection, preprocessing, feature engineering, model selection, training, evaluation, and deployment.

Traditionally, this entire process is managed through extensive programming, requiring deep expertise in languages like Python and its associated libraries (e.g., Scikit-learn, Pandas, TensorFlow). While powerful, this code-centric approach presents several challenges, particularly for beginners, interdisciplinary teams, and even experienced practitioners aiming for rapid prototyping.

### 1.1 MOTIVATION

The motivation for this project stems from three primary challenges in the contemporary ML landscape:

1. High Barrier to Entry: The reliance on complex code creates a steep learning curve for students, domain experts, and business analysts who may understand the data and objectives but lack the programming skills to implement ML solutions.
2. Lack of Standardization and Reproducibility: Code-based workflows can be difficult to share, understand, and reproduce. Different developers may structure their code differently, and tracking the exact parameters and datasets used for a specific model result can become a significant challenge. This issue is a major bottleneck in both academic research and enterprise-level MLOps (Machine Learning Operations).
3. Slow Prototyping Cycle: Writing, debugging, and refactoring code for each new experiment is time-consuming. A significant amount of effort is spent on boilerplate code rather than on the core logic of the ML model, slowing down the innovation cycle.

Visual programming paradigms have successfully lowered the barrier to entry in many other domains. This project is motivated by the belief that a well-designed visual tool can bring similar benefits to the ML workflow, making it more accessible, intuitive, and efficient without sacrificing the power of the underlying code.

### 1.2 PROBLEM STATEMENT

Existing machine learning workflows are often fragmented, code-intensive, and difficult to manage, leading to inefficiencies in model development, poor reproducibility, and a high barrier to entry for non-programmers. There is a need for an integrated, user-friendly platform that allows for the rapid visual construction of ML pipelines, while also providing robust tools for experiment tracking, version control, and seamless translation of visual designs into production-ready code.

This project aims to solve this problem by developing DominoML, a web-based application that enables users to visually build, validate, and manage ML pipelines through a drag-and-drop interface, with built-in support for versioning and automatic Python code generation.

### 1.3 OBJECTIVE

To address the problem statement, the project is guided by the following primary objectives:

1. To Develop an Intuitive Visual Pipeline Builder: Create a web-based, drag-and-drop canvas where users can visually construct ML workflows by selecting and connecting predefined components.
2. To Provide a Comprehensive Component Library: Curate and implement a library of essential ML components covering data loading, preprocessing, model algorithms (classification, regression, etc.), and evaluation.
3. To Implement Real-time Validation: Integrate a validation engine that provides immediate feedback to the user about the structural and logical integrity of the constructed pipeline (e.g., ensuring correct data types are passed between nodes).
4. To Enable Automatic Code Generation: Develop a robust mechanism to translate the visual pipeline into clean, executable, and well-documented Python code using standard libraries like Scikit-learn and Pandas.
5. To Implement User and Model Management: Build a secure user authentication system and a database backend to allow users to save, load, and manage their created pipelines.
6. To Integrate an Undo/Redo System: Enhance user experience by implementing a history management system that allows users to undo and redo actions on the canvas, with persistence across sessions.
7. To Develop a Pipeline Versioning System: Create a full-fledged version control system for saved pipelines, enabling users to track changes, compare different versions, store experiment metrics, and manage the model lifecycle.

### 1.4 SCOPE AND LIMITATIONS

#### *1.4.1 In-Scope Features*

Platform: A responsive, browser-based web application.

Core Functionality: Visual drag-and-drop interface for building pipelines.

Components: Focus on classical ML algorithms available in Scikit-learn.

Data Sources: Support for CSV file uploads and pre-loaded sample datasets.

User Management: Secure user registration, login, and session management.

Persistence: Saving and loading user-created pipelines to a SQLite database.

Productivity Tools: Undo/Redo functionality and a gallery of pre-built templates.

Versioning: Full version tracking with metadata, metric storage, and release tagging.

Export: Generation of Python scripts and export/import of pipelines as JSON.

#### *1.4.2 Out of Scope Features*

Deep Learning Frameworks: The component library will not include components for deep learning frameworks like TensorFlow or PyTorch.

Big Data Technologies: The application is not designed to integrate with big data platforms like Apache Spark.

Real-time Model Deployment: The project focuses on model creation and code generation, not on deploying models as live API endpoints.

Automated Hyperparameter Tuning: While parameters can be set manually, the system will not perform automated hyperparameter optimization (e.g., Grid Search, Bayesian Optimization).

Real-time Collaboration: Multiple users cannot edit the same pipeline simultaneously.

Interactive Guided Learning: New users can follow on-screen prompts to learn how to build models.

### 1.5 APPLICATION AND TARGET AUDIENCE

DominoML is designed to serve a diverse range of users within the data science and software engineering ecosystems:

* Students and Educators: Provides an interactive, visual environment for learning and teaching ML concepts without getting bogged down by complex code. It helps in understanding the flow and logic of different model architectures.
* Data Analysts and Business Intelligence Professionals: Empowers analysts to build and test predictive models on their data, enabling them to move from descriptive to predictive analytics without requiring extensive programming knowledge.
* Machine Learning Engineers and Data Scientists: Acts as a rapid prototyping tool to quickly build and iterate on baseline models. The code generation feature allows them to start with a solid foundation and then move to a code-based environment for fine-tuning and optimization.
* Software Developers: Offers a quick way to integrate ML models into applications. They can use the tool to design the pipeline and generate the necessary Python script to be included in their backend services.

### 1.6 REPORT ORGANIZATION

This project report is structured into eight chapters to provide a comprehensive overview of the work undertaken.

1. Chapter 1: Introduction provides the motivation, problem statement, objectives, and scope of the project.
2. Chapter 2: Literature Survey and Background Study reviews existing tools in the market, discusses the theoretical underpinnings of MLOps, and explains the rationale behind the chosen technology stack.
3. Chapter 3: System Requirements Specification formally lists the functional and non-functional requirements that guided the development of the application.
4. Chapter 4: System Design and Architecture details the high-level architecture, database schema, UML diagrams, and the UI/UX design of the platform.
5. Chapter 5: Implementation Details offers an in-depth look at the project's structure and the implementation of its core features, from the backend APIs to the frontend canvas logic.
6. Chapter 6: System Testing describes the testing strategies, methodologies, and sample test cases used to ensure the application's correctness, reliability, and usability.
7. Chapter 7: Results and Snapshots presents the final output of the project through screenshots and examples of generated code.
8. Chapter 8: Conclusion and Future Scope summarizes the project's achievements, acknowledges its limitations, and proposes directions for future development.

## CHAPTER 2: LITERATURE SURVEY AND BACKGROUND STUDY

The development of DominoML is not occurring in a vacuum. It is a direct response to several converging trends in software development, workflow automation, and machine learning education. This chapter explores these trends, analyzes existing tools, and justifies the technological choices made for this project.

### 2.1 THE PARADIGM SHIFT: THE RISE OF VISUAL WORKFLOW AUTOMATION

In the post-2023, AI-driven landscape, speed and efficiency are paramount. The software industry has witnessed a dramatic shift towards no-code and low-code platforms. Tools like **n8n, Zapier, and Make.com** have fundamentally changed how businesses automate processes. They replace complex scripting with intuitive, visual, drag-and-drop interfaces, empowering non-developers to build sophisticated workflows.

This paradigm has proven that visually connecting logical blocks is a highly effective way to design, understand, and manage complex systems. The core lesson is that abstraction, when done right, does not diminish power; it democratizes it and accelerates innovation. DominoML applies this same philosophy to the domain of machine learning.

### 2.2 THE "ABSTRACTION GAP" IN MACHINE LEARNING PRACTICE

The traditional path to learning machine learning is steep and rooted in mathematics—linear algebra, calculus, statistics, and probability theory. While fundamental, this approach is time-consuming and often presents a significant barrier to entry. Implementing algorithms like Support Vector Machines or Gradient Boosting from scratch is a valuable academic exercise but is impractical for most real-world applications.

This has led to a common practice we term the **"Library Leap."** Aspiring practitioners often bypass the theoretical foundations and jump directly to using high-level libraries like Scikit-learn. They learn the syntax—model.fit(X, y)—but may not grasp the underlying concepts. This creates a critical "abstraction gap":

1. **ML as a "Black Box":** Without understanding the flow, users treat models as magical black boxes. They may not know why a StandardScaler is necessary before a PCA, or why one-hot encoding is used for categorical features.
2. **Ineffective Debugging:** When a model performs poorly, it's difficult to diagnose the problem without a conceptual map of the entire pipeline. Is the issue in data cleaning, feature engineering, or model selection?
3. **Inhibited Experimentation:** The cognitive overhead of writing and rewriting Python scripts for each minor change in the pipeline can stifle creative experimentation.

DominoML is explicitly designed to bridge this gap. It provides a visual middle ground that illustrates the **logic and flow** of an ML pipeline—how data is transformed and passed from one step to the next—without requiring the user to perform complex mathematical derivations or write boilerplate code from scratch. It makes the connections between library functions tangible.

### 2.3 A CRITICAL REVIEW OF EXISTING VISUAL ML PLATFORMS

Several visual ML tools exist, each with its own strengths and target audience. A candid assessment reveals the specific niche DominoML aims to fill.

#### *2.3.1 Desktop-Based Analytics Workbenches (KNIME, Orange)*

* **KNIME Analytics Platform** and **Orange Data Mining** are powerful, open-source, desktop-based applications. They have extensive libraries of nodes and are well-regarded in the academic and research communities.
* **Honest Assessment:** While powerful, their primary limitation is their platform. As desktop applications, they lack the accessibility and collaborative potential of modern web-based tools. Their user interfaces, though functional, can feel dated and less intuitive compared to contemporary SaaS products. Furthermore, their focus is often on data analysis within the tool itself, rather than on generating clean, idiomatic, and educational Python code that a developer can easily integrate into another project.

#### *2.3.2 Enterprise-Grade Cloud Platforms (Azure ML, Vertex AI)*

* **Azure Machine Learning Designer** and **Google Cloud's Vertex AI Pipelines** are state-of-the-art, enterprise-focused solutions. They offer immense scalability, integration with a vast ecosystem of cloud services, and are built for production-level MLOps.
* **Honest Assessment:** Their power comes at a cost—both literally and figuratively. These platforms introduce significant complexity, are tied to a specific vendor's ecosystem (vendor lock-in), and can be expensive to use. They are overkill for students, individual developers, or small teams focused on rapid prototyping and learning. Their primary goal is deployment within their cloud, not creating portable, framework-agnostic Python scripts for educational purposes.

#### *2.3.3 Positioning DominoML*

DominoML carves its niche by deliberately avoiding the heavyweight nature of both categories. It is designed to be:

1. **Lightweight and Accessible:** A pure web application with no installation required, running on a simple Flask/SQLite stack.
2. **Education-First:** Its primary goal is to demystify ML pipelines and generate clean, readable Python code that serves as a learning tool.
3. **Prototyping-Focused:** Optimized for speed and ease of use, allowing for rapid iteration of model ideas.
4. **Open and Unopinionated:** It is open-source and generates standard Python code, freeing the user from any platform lock-in.

### 2.4 JUSTIFICATION OF FOUNDATIONAL TECHNOLOGIES

The technology stack was chosen to align with these goals of simplicity, accessibility, and maintainability.

1. **Backend (Flask):** Flask is a micro-framework for Python. Unlike Django, it is minimal and unopinionated, providing maximum flexibility. This was ideal for building a custom application with a specific set of requirements, avoiding the overhead of features not needed for this project.
2. **Database (SQLAlchemy with SQLite):** SQLite is a serverless, self-contained database engine, which is perfect for development and small-to-medium scale deployment as it requires zero configuration. SQLAlchemy, as an Object-Relational Mapper (ORM), provides a high-level abstraction over the database, allowing for clean, Pythonic database interactions and making it easy to switch to a more powerful database like PostgreSQL in the future if needed.
3. **Frontend (Vanilla JavaScript):** In an era dominated by complex frameworks like React and Vue, the choice of vanilla JavaScript was deliberate. The core challenge of the frontend is managing the state of a complex, interactive canvas. Using vanilla JS forces a deeper understanding of the DOM and provides unparalleled performance and control, avoiding the overhead and potential limitations of a framework's virtual DOM. It also keeps the application lightweight and dependency-free.

## **CHAPTER 3: SYSTEM REQUIREMENTS SPECIFICATION**

This chapter defines the specific requirements for the DominoML application. These requirements are categorized into functional (what the system does) and non-functional (how the system does it) aspects, along with the necessary hardware and software environments.

### 3.1 FUNCTIONAL REQUIREMENTS

***FR-1: User Authentication***

1. **FR-1.1:** The system shall allow new users to register for an account using a unique username, a valid email address, and a password.
2. **FR-1.2:** The system shall allow registered users to log in using their credentials.
3. **FR-1.3:** The system shall manage user sessions, ensuring that a user remains logged in across different pages.
4. **FR-1.4:** The system shall provide a mechanism for users to log out.

***FR-2: Visual Pipeline Builder***

1. **FR-2.1:** The system shall display a library of ML components, categorized into Data Sources, Preprocessing, Models, and Evaluation.
2. **FR-2.2:** Users shall be able to drag components from the library and drop them onto a canvas.
3. **FR-2.3:** Users shall be able to create connections (edges) between compatible input/output nodes of the components on the canvas.
4. **FR-2.4:** The system shall prevent the creation of connections between incompatible nodes (e.g., connecting a text output to a numerical input).
5. **FR-2.5:** Users shall be able to select, move, and delete components and connections on the canvas.

***FR-3: Component Configuration***

1. **FR-3.1:** When a user selects a component on the canvas, the system shall display a properties panel with its configurable parameters.
2. **FR-3.2:** Users shall be able to modify the parameters of a component (e.g., change the number of neighbors in a k-NN model).

***FR-4: Pipeline Management***

1. **FR-4.1:** Logged-in users shall be able to save their current pipeline to the database with a unique name and description.
2. **FR-4.2:** Users shall be able to view a gallery of their previously saved models.
3. **FR-4.3:** Users shall be able to load any of their saved models back onto the canvas.
4. **FR-4.4:** The system shall provide a library of pre-built template pipelines that any user can load.
5. **FR-4.5:** The system shall allow users to export their pipeline's structure as a JSON file and import a pipeline from a valid JSON file.

***FR-5: Core Functionality***

1. **FR-5.1:** The system shall provide real-time validation of the pipeline structure, highlighting any errors such as disconnected but required nodes.
2. **FR-5.2:** The system shall be able to generate a complete, executable Python script based on the visual pipeline on the canvas.
3. **FR-5.3:** The system shall provide an undo/redo mechanism for up to 50 recent actions on the canvas, with keyboard shortcuts (Ctrl+Z, Ctrl+Shift+Z).

***FR-6: Pipeline Versioning***

1. **FR-6.1:** Users shall be able to create a new, immutable version of a saved pipeline, with a version name, description, and tags.
2. **FR-6.2:** The system shall display a visual timeline or list of all versions for a given pipeline.
3. **FR-6.3:** Users shall be able to load any previous version of a pipeline onto the canvas.
4. **FR-6.4:** The system shall provide an API endpoint to store key-value metrics (e.g., accuracy, F1-score) associated with a specific pipeline version.

### 3.2 NON-FUNCTIONAL REQUIREMENTS

***NFR-1: Performance***

* **NFR-1.1:** The web application pages shall load in under 3 seconds on a standard broadband connection.
* **NFR-1.2:** All canvas interactions (drag, drop, connect) shall provide immediate visual feedback (< 100ms latency).
* **NFR-1.3:** Code generation for a moderately complex pipeline (10-15 nodes) shall complete in under 2 seconds.

***NFR-2: Usability***

1. **NFR-2.1:** The user interface shall be intuitive and self-explanatory for users with a basic understanding of flowchart diagrams.
2. **NFR-2.2:** The application shall be responsive and usable on modern web browsers on desktop, tablet, and mobile devices.
3. **NFR-2.3:** The system shall include a theme toggle for switching between a light and a dark mode.

***NFR-3: Security***

1. **NFR-3.1:** All user passwords shall be securely hashed before being stored in the database.
2. **NFR-3.2:** The application shall be protected against common web vulnerabilities, including Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF).
3. **NFR-3.3:** User data shall be isolated; a user must not be able to view or modify pipelines belonging to another user.

***NFR-4: Maintainability & Scalability***

1. **NFR-4.1:** The backend and frontend code shall be modular, well-documented, and follow standard coding practices (PEP 8 for Python).
2. **NFR-4.2:** The system architecture shall be decoupled (frontend, backend, database), allowing for independent development and updates.
3. **NFR-4.3:** The application should be capable of supporting at least 50 concurrent users without significant performance degradation.

### 3.3 HARDWARE REQUIREMENTS (FOR HOSTING)

1. **CPU:** 1 vCPU (minimum)
2. **RAM:** 512 MB (minimum), 1 GB (recommended)
3. **Storage:** 5 GB of free disk space for the application, database, and logs.

### 3.4 SOFTWARE REQUIREMENTS

#### *3.4.1 Development/Deployment Environment*

1. **Operating System:** Any modern Linux, Windows, or macOS distribution.
2. **Programming Language:** Python 3.8 or higher.
3. **Web Server (for production):** A WSGI server like Gunicorn.
4. **Database:** SQLite 3.x (for development), PostgreSQL (recommended for production).

#### *3.4.2 Client-Side*

1. **Web Browser:** Latest version of Google Chrome, Mozilla Firefox, Microsoft Edge, or Safari.
2. JavaScript must be enabled.

### 3.5 USER CHARACTERISTICS

The intended users of the system are expected to have basic computer literacy and familiarity with web applications. While no programming knowledge is required to use the visual builder, an understanding of fundamental machine learning concepts (e.g., what is a dataset, what is a model) is assumed. Users who wish to utilize the generated code are expected to have a working knowledge of Python.

## **CHAPTER 4: SYSTEM DESIGN AND ARCHITECTURE**

Effective system design is the blueprint for a successful software project. This chapter details the architectural decisions, database structure, and user interface design that form the foundation of DominoML. The design prioritizes modularity, maintainability, and a clear separation of concerns between the backend logic and the frontend user experience.

### 4.1 HIGH-LEVEL SYSTEM ARCHITECTURE

DominoML is architected as a classic three-tier web application, a proven model for building robust and scalable systems. The three tiers are:

1. **Presentation Tier (Frontend):** This is the client-side of the application, running entirely within the user's web browser. It is built with HTML, custom CSS, and vanilla JavaScript. Its sole responsibilities are to render the user interface, handle user interactions on the canvas, manage the visual state of the pipeline, and communicate with the backend via API calls. This decoupling ensures that the UI can be updated or even completely replaced without affecting the backend logic.
2. **Logic Tier (Backend):** This is the server-side of the application, powered by a Flask web server. It is the brain of the system, responsible for:
   * Handling user authentication and session management.
   * Serving the static frontend files (HTML, CSS, JS).
   * Providing a RESTful API for all Create, Read, Update, Delete (CRUD) operations on pipelines and their versions.
   * Executing the core business logic, including pipeline validation and Python code generation.
3. **Data Tier (Database):** This tier is responsible for the persistent storage of all application data. It consists of a SQLite database managed by the SQLAlchemy ORM. Data stored includes user account information, saved pipeline structures, and all associated versioning data and metrics.

This architecture is illustrated in the diagram below.

**Figure 4.1: High-Level System Architecture**

codeCode

+-----------------------------+

| User's Browser |

|-----------------------------|

| Frontend (Vanilla JS) |

| - Renders UI (Canvas, Panels) |

| - Handles User Input |

| - Manages Visual State |

+--------------+--------------+

|

(HTTPS Requests / REST API)

|

+--------------v--------------+

| Backend (Flask) |

|-----------------------------|

| - Authentication (Flask-Login)|

| - REST API Routes (api.py) |

| - Business Logic |

| - Code Generation |

| - Validation |

| - Versioning Logic |

+--------------+--------------+

|

(SQLAlchemy ORM Queries)

|

+--------------v--------------+

| Database (SQLite) |

|-----------------------------|

| - Users Table |

| - Saved\_Models Table |

| - Pipeline\_Versions Table |

+-----------------------------+

### 4.2 TECHNOLOGY STACK

The technology stack was chosen for its simplicity, maturity, and alignment with the project's goals.

1. **Backend:**
   * **Framework:** Flask 3.0
   * **Database ORM:** SQLAlchemy
   * **Authentication:** Flask-Login
   * **Forms:** Flask-WTF
2. **Frontend:**
   * **Language:** Vanilla JavaScript (ECMAScript 6+)
   * **Templating:** Jinja2 (server-side rendering for initial page load)
   * **Styling:** Custom CSS (replicating a modern utility-first design system)
3. **Database:**
   * **Development:** SQLite
   * **Production Recommendation:** PostgreSQL
4. **Development & Deployment:**
   * **Environment:** Python Virtual Environments (venv)
   * **Server:** Werkzeug (development), Gunicorn (production)

### 4.3 DATABASE DESIGN

The database is the heart of the application's persistence layer. The schema is designed to be relational and normalized to ensure data integrity and efficiency.

#### *4.3.1 Entity-Relationship (ER) Diagram*

The ER diagram below shows the core entities and their relationships. A User can have many SavedModels. Each SavedModel (representing a pipeline) can have many PipelineVersions. Each version can have its own associated Metrics.

**Figure 4.2: Entity-Relationship Diagram for DominoML Database**

codeCode

+----------+ 1..\* +-------------+ 1..\* +------------------+

| User |----------------| SavedModel |----------------| PipelineVersion |

+----------+ +-------------+ +------------------+

| user\_id (PK) | | model\_id (PK)| | version\_id (PK) |

| username | | name | | model\_id (FK) |

| email | | description | | version\_name |

| password | | user\_id (FK)| | description |

| ... | | json\_data | | json\_data |

+----------+ | ... | | created\_at |

+-------------+ | is\_active |

+---------+--------+

| 1..\*

+---------v--------+

| Metric |

+------------------+

| metric\_id (PK) |

| version\_id (FK) |

| key |

| value |

+------------------+

#### *4.3.2 Schema Definition*

1. **User:** Stores user account details. The password field stores a secure hash, not the plaintext password.
   * id (Integer, Primary Key)
   * username (String, Unique)
   * email (String, Unique)
   * password\_hash (String)
2. **SavedModel:** Represents a single, editable ML pipeline. This is the "working copy."
   * id (Integer, Primary Key)
   * name (String)
   * description (Text)
   * graph\_json (Text) - Stores the current state of the pipeline canvas as a JSON string.
   * user\_id (Integer, Foreign Key to User.id)
3. **PipelineVersion:** Represents an immutable snapshot (a version) of a SavedModel.
   * id (Integer, Primary Key)
   * model\_id (Integer, Foreign Key to SavedModel.id)
   * version\_name (String)
   * description (Text)
   * graph\_json (Text) - Stores the pipeline state for this specific version.
   * created\_at (DateTime) - Timestamp of when the version was created.
   * is\_active (Boolean) - A flag to mark a version for "production."
4. **Metric:** Stores key-value metrics associated with a specific version, used for experiment tracking.
   * id (Integer, Primary Key)
   * version\_id (Integer, Foreign Key to PipelineVersion.id)
   * key (String, e.g., "accuracy", "f1\_score")
   * value (Float)

### 4.4 DESIGN FRAMEWORKS

#### *4.4.1 Use Case Diagram*

The Use Case Diagram illustrates the interactions between a Registered User and the DominoML system.

**Figure 4.3: Use Case Diagram for a Registered User**

codeCode

+-------------------------------------+

| DominoML System |

| |

+-------------------+ | (Manage Account) |

| Registered User |---|--> (Build Pipeline) |

+-------------------+ | ^ extends (Load Template) |

| | ^ extends (Load Saved Model) |

| | |

|------------|--> (Save Pipeline) |

| | |

|------------|--> (Generate Python Code) |

| | |

|------------|--> (Manage Versions) |

| | ^ includes (Store Metrics) |

| | |

|------------|--> (Undo/Redo Actions) |

| | |

+-------------------------------------+

(Unregistered User can only access the landing page and registration form.)

#### *4.4.2 Data Flow Diagrams (DFD)*

**Figure 4.4: DFD Level 0 (Context Diagram)**This diagram shows the overall system as a single process and its interaction with external entities.

codeCode

+-------------+ Pipeline Data +-----------------+

| Registered |<----------------------->| |

| User | | 0. DominoML |

+-------------+ Auth Credentials | System |

|-------------------------------->| |

+-----------------+

**Figure 4.5: DFD Level 1 (Process Breakdown)**This diagram breaks down the main system into its core processes.

codeCode

+------------------+

| Pipeline JSON |

+--------+---------+

|

+------+ Canvas Actions +-------------v-------------+ Saved Pipeline

| User |------------------->| 1. Manage Canvas State |----------------->+

+------+ +---------------------------+ |

^ ^ | v |

| | Validation Errors | |

| Rendered Canvas | | |

| | Pipeline JSON | +--------------v---+

| | +-->| 3. Manage Models |

| | +-------------->| & Versions |<->[D1: Pipelines DB]

| | | +----------------+

| Auth Credentials +--v-------------------------+ ^

+------------------------| 2. Authenticate User |<--------->[D2: Users DB]

+---------------------------+ |

| +---------v--------+

Pipeline JSON & Config ---->| 4. Generate Code |

+------------------+

|

Python Code |

v

(User)

### 4.5 FRONTEND DESIGN AND UI/UX

The user interface (UI) and user experience (UX) are critical for a visual tool. The design philosophy is centered around minimalism, clarity, and efficiency, ensuring the user can focus on building pipelines without distraction.

#### *4.5.1 The Builder Interface Layout*

The main builder page is divided into three primary sections, a layout common in modern creative software.

**Figure 4.6: The Main Builder Interface Layout**

codeCode

+-----------------------------------------------------------------------------+

| Toolbar (Save, Versions, Validate, Code Gen, Undo/Redo, Theme) |

+----------------------+------------------------------------+-----------------+

| | | |

| Component Library | | Property Panel |

| (Sidebar) | | (Sidebar) |

| | | |

| - Data Sources | | - Component Name|

| - Preprocessing | Main Canvas | - Parameters |

| - Models | (Drag & Drop Area) | - Param 1 |

| - Evaluation | | - Param 2 |

| | | - ... |

| | | |

| | | |

+----------------------+------------------------------------+-----------------+

| Footer (Status Bar: Validation messages, last saved time) |

+-----------------------------------------------------------------------------+

1. **Component Library (Left):** A collapsible sidebar containing all available ML nodes, grouped by category. Users drag items from here onto the canvas.
2. **Main Canvas (Center):** The primary interactive area where the pipeline is constructed. It supports panning and zooming for large workflows.
3. **Property Panel (Right):** A context-aware sidebar. When no node is selected, it shows global pipeline settings. When a node is selected, it displays that node's specific parameters for editing.

#### *4.5.2 Design System and Color Palette*

A consistent design system is used to improve usability and convey information visually.

1. **Color Palette:**
   * **Light Mode:** Uses a clean white background with dark grey text and vibrant blue/purple accents for interactive elements.
   * **Dark Mode:** Employs a dark charcoal background (#1a202c) with light grey text to reduce eye strain, maintaining a high-contrast, accessible theme.
   * **Component Colors:** Each component category has a distinct color code to make pipelines instantly readable.
2. **Typography & Icons:** A clean, sans-serif font is used for readability. Icons from the **Lucide Icons** library are used throughout the UI for their clarity and minimalist aesthetic. This consistent visual language helps users quickly identify actions and components.

## **CHAPTER 5: IMPLEMENTATION DETAILS**

This chapter provides a detailed examination of the implementation of DominoML, translating the design specifications from Chapter 4 into tangible software components. It covers the project's file structure, the core logic of both the backend and frontend, and the specific implementation of key features like code generation and pipeline versioning.

### 5.1 PROJECT STRUCTURE

A well-organized project structure is crucial for maintainability. DominoML follows the Flask Application Factory pattern, which promotes modularity and scalability.

**Figure 5.1: DominoML Project Directory Structure**

codeCode

dominoML-flask/

├── app/

│ ├── \_\_init\_\_.py # Application factory (create\_app)

│ ├── models.py # SQLAlchemy database models

│ ├── forms.py # WTForms classes for auth

│ ├── data/

│ │ ├── ml\_components.json # Definitions for all ML nodes

│ │ └── ml\_templates.json # Pre-built pipeline templates

│ ├── routes/

│ │ ├── main.py # Routes for main pages (landing, builder)

│ │ ├── auth.py # Authentication routes (login, signup)

│ │ └── api.py # All REST API endpoints

│ ├── static/

│ │ ├── css/ # CSS files

│ │ └── js/ # JavaScript files (canvas.js, history.js, etc.)

│ └── templates/

│ ├── base.html # Master template with header/footer

│ └── ... # Other HTML templates

├── config.py # Configuration settings (secret key, DB URI)

├── run.py # Main entry point to run the application

└── requirements.txt # Python dependencies

### 5.2 BACKEND IMPLEMENTATION (FLASK)

The backend is the authoritative core of the application, handling all business logic and data persistence.

#### *5.2.1 Application Factory and Configuration*

The create\_app() function in \_\_init\_\_.py serves as the application factory. This pattern is a Flask best practice that allows for the creation of multiple application instances with different configurations (e.g., for production, development, or testing). It initializes the database (SQLAlchemy), login manager (Flask-Login), and registers the route blueprints.

Configuration variables, such as the SECRET\_KEY and DATABASE\_URL, are loaded from a config.py file, which in turn reads from environment variables. This separation prevents hardcoding sensitive information directly into the codebase.

#### *5.2.2 Database Models*

This file defines the application's database schema using SQLAlchemy's ORM. Each class (User, SavedModel, PipelineVersion) maps directly to a table in the database.

A key implementation detail is within the User model, which includes methods for password management:

codePython

from werkzeug.security import generate\_password\_hash, check\_password\_hash

class User(db.Model, UserMixin):

# ... column definitions ...

def set\_password(self, password):

self.password\_hash = generate\_password\_hash(password)

def check\_password(self, password):

return check\_password\_hash(self.password\_hash, password)

This ensures that raw passwords are never stored, only their secure hashes.

#### *5.2.3 Authentication Module*

This blueprint handles all user authentication logic. It uses Flask-Login for session management and Flask-WTF for secure form handling, which includes built-in CSRF protection. The routes define the logic for rendering the login/signup forms, validating user input, creating new users, and managing login sessions.

#### *5.2.4 API Endpoints*

This is the most critical backend module, as it provides the interface for the frontend to interact with the server's data and logic. All communication is done via JSON.

Key endpoints include:

1. GET /api/components: Loads and returns the ml\_components.json file, populating the frontend's component library.
2. POST /api/models/save: Receives a pipeline's JSON data from the frontend. It verifies the user is authenticated and then creates or updates the corresponding SavedModel record in the database.
3. GET /api/models/<int:model\_id>: Retrieves a specific pipeline for a logged-in user.
4. POST /api/versions/create: Creates a new, immutable PipelineVersion from the current state of a SavedModel.
5. POST /api/generate-code: Receives a pipeline's JSON data and passes it to the code generation utility. Returns the generated Python script as a JSON response.

### 5.3 FRONTEND IMPLEMENTATION (VANILLA JAVASCRIPT)

The frontend is a single-page application (SPA) experience within the /builder route. The JavaScript is modularized to handle different aspects of the UI.

#### *5.3.1 The Canvas Logic*

This is the core of the frontend, responsible for all visual rendering and interaction on the main canvas.

1. **State Management:** It maintains a JavaScript object that represents the entire state of the canvas, including all nodes (with their positions and parameters) and edges (connections). Every user action modifies this state object.
2. **Rendering Loop:** A simple rendering function is called whenever the state changes. This function clears the canvas and redraws all nodes and edges from the current state object. This approach, while less sophisticated than a virtual DOM, is highly performant for this specific use case.
3. **Event Handling:** The script listens for mouse events (mousedown, mousemove, mouseup) to handle dragging nodes, creating connections, and selecting elements.

#### *5.3.2 Component and Properties Management*

1. components.js: Fetches the component definitions from the /api/components endpoint on page load. It then dynamically generates the HTML for the component library sidebar.
2. properties.js: Is responsible for the right-hand properties panel. When a node is selected on the canvas, canvas.js emits a custom event. properties.js listens for this event and dynamically builds an HTML form based on the selected node's configurable parameters (as defined in its JSON specification). When a parameter is changed, it updates the main state object in canvas.js.

#### *5.3.3 Undo/Redo System*

This module implements the undo/redo functionality, a critical UX feature.

1. **History Stack:** It maintains two arrays: undoStack and redoStack.
2. **State Snapshots:** Before any significant action is performed on the canvas (e.g., adding a node, deleting an edge), canvas.js calls a function in history.js to push a deep copy of the *current* canvas state onto the undoStack.
3. **Undo Operation:** When the user triggers an undo, the current state is pushed to the redoStack, and the last state is popped from the undoStack and loaded as the new canvas state.
4. **Redo Operation:** The opposite of undo; a state is moved from the redoStack back to the undoStack.
5. **Persistence:** The last 10 states from the undoStack are periodically saved to the browser's localStorage, allowing the user to recover their recent actions even after a page reload.

#### *5.3.4 Version Management UI*

This script manages the modal window for pipeline versioning. It makes API calls to fetch all versions for the current model, renders them in a timeline view, and handles events for creating new versions or loading existing ones.

### 5.4 CORE FEATURE IMPLEMENTATION

#### *5.4.1 Code Generation Logic*

This is a standalone Python utility that encapsulates the logic for translating the pipeline JSON into Python code. This separation ensures the logic can be tested independently of the Flask application.

The process works as follows:

1. **Topological Sort:** The generator first receives the list of nodes and edges. It performs a topological sort on this directed acyclic graph (DAG) to determine the correct order of execution. This ensures that a node is only processed after all its inputs are available.
2. **Node Traversal:** It iterates through the sorted nodes one by one.
3. **Code Template Mapping:** For each node type (e.g., CsvDataSource, StandardScaler, LogisticRegression), the generator has a corresponding Python code template.
4. **Dynamic Code Assembly:** It dynamically fills in these templates using the parameters specified by the user in the properties panel. It also manages variable names, ensuring the output of one node is correctly passed as the input to the next.
5. **Final Script Generation:** The generator assembles the code blocks for library imports, the pipeline steps, and a main execution block, resulting in a single, clean, and executable Python script.

#### *5.4.2 Real-time Validation*

Validation is performed on both the frontend and backend.

1. **Frontend Validation (in**  Provides instant feedback. It checks for basic structural issues like:
   * Unconnected but required input nodes.
   * Incompatible data types between connections (e.g., trying to connect a trained model to a data input).  
     This logic is simple and visual, highlighting problematic nodes in red.
2. **Backend Validation (in**  Before saving a model or generating code, a more thorough validation is run on the server. This serves as a security and integrity check, ensuring the JSON data submitted by the client is well-formed and logically sound.

#### *5.4.3 Pipeline Versioning and Metric Tracking*

The implementation of versioning is primarily an API and database concern.

1. When a user clicks "Create New Version," the frontend POSTs the current SavedModel ID and version details to /api/versions/create.
2. The backend retrieves the graph\_json from the parent SavedModel, creates a new PipelineVersion record with this data, and saves it. This effectively creates an immutable snapshot.
3. Storing metrics is handled by a separate endpoint (POST /api/versions/<id>/metrics). This allows for programmatic tracking of experiments, where a training script could, for example, post its resulting accuracy back to the specific version it was generated from.

## **CHAPTER 6: SYSTEM TESTING**

System testing is a critical phase that validates the application's functionality, reliability, and usability. It is not merely about finding bugs but about ensuring the software meets the specified requirements and provides a positive user experience. This chapter outlines the testing strategies and methodologies employed for DominoML, focusing on a pragmatic approach that prioritizes critical user workflows.

### 6.1 TESTING OBJECTIVES

The primary objectives of the testing phase were:

1. **To Verify Functional Correctness:** Ensure that all features defined in the System Requirements Specification (Chapter 3) work as intended. This includes everything from user login to code generation.
2. **To Ensure a Smooth User Experience:** Identify and address any UI/UX issues, such as unresponsive elements, confusing layouts, or broken workflows.
3. **To Validate Data Integrity:** Confirm that data is being correctly and securely saved, retrieved, and managed in the database, with proper user-based access control.
4. **To Confirm System Stability:** Test the application's behavior under typical usage patterns to identify any critical errors or crashes.
5. **To Validate the Generated Code:** The most critical objective—ensure the Python code generated by the application is syntactically correct, logically sound, and produces the expected results when executed.

### 6.2 TESTING METHODOLOGIES

A multi-layered testing approach was adopted, combining manual and automated techniques. Given the highly interactive and visual nature of the canvas, manual testing was indispensable.

#### *6.2.1 Unit Testing (Backend)*

1. **Approach:** Unit tests were written for critical, isolated pieces of backend logic using Python's built-in unittest framework. The focus was on "pure" functions that did not depend on the database or external requests.
2. **Key Areas Covered:**
   * **Code Generation (** The most heavily unit-tested component. Tests were created with sample pipeline JSON structures to verify that the topological sort was correct and that the generated code for individual nodes was accurate.
   * **Models (** Tests to ensure the password hashing and verification methods on the User model worked correctly.
3. **Honest Assessment:** Comprehensive unit testing of every single function was not feasible within the project's scope. The strategy was to target the most complex and critical logic—the code generator—where an error would have the most significant impact.

#### *6.2.2 Integration Testing (Backend)*

1. **Approach:** Integration tests focused on verifying the interactions between different parts of the backend, particularly between the API routes and the database. These tests were conducted using a temporary, in-memory SQLite database to ensure a clean state for each test run.
2. **Key Areas Covered:**
   * **API Endpoints (** Tests were written to simulate HTTP requests to the API endpoints (e.g., saving a model, creating a version) and assert that the database state changed correctly.
   * **Authentication Flow:** Testing the entire registration-login-logout cycle to ensure session management was working as expected.

#### *6.2.3 Manual End-to-End System Testing*

1. **Approach:** This was the most significant part of the testing effort. It involved manually performing user workflows from start to finish, simulating real-world usage. A series of test cases were designed to cover the main "happy paths" as well as common edge cases.
2. **Key Workflows Tested:**
   1. **New User Onboarding:** Register a new account, log in, and verify access to the builder.
   2. **Full Pipeline Creation:**
      * Drag and drop a data source, a preprocessing step, a model, and an evaluation node.
      * Connect them in a logical sequence.
      * Configure parameters for each node.
      * Save the pipeline.
      * Log out and log back in to ensure the pipeline can be successfully reloaded.
   3. **Code Generation and Validation:**
      * Create a complete, valid pipeline.
      * Click the "Generate Code" button.
      * Copy the generated Python script.
      * Run the script in a separate local Python environment.
      * Verify that the script runs without errors and produces a sensible output (e.g., an accuracy score).
   4. **Versioning Workflow:**
      * Save a pipeline (Version A).
      * Create a named version "v1.0".
      * Modify the pipeline on the canvas (e.g., change a model parameter) and save it (Version B).
      * Create a new named version "v1.1".
      * Load version "v1.0" and verify the canvas reverts to the state of Version A.
   5. **Undo/Redo Functionality:**
      * Perform a sequence of actions (add node, connect, delete node).
      * Use the undo button/shortcut to reverse each action.
      * Use the redo button/shortcut to re-apply each action.

#### *6.2.4 User Acceptance Testing (UAT)*

* **Approach:** Informal UAT was conducted by asking peers (fellow students with some ML knowledge) to use the application without a strict script. They were given a simple goal (e.g., "Try to build a pipeline that predicts house prices") and were observed.
* **Key Feedback Gathered:** This phase was less about finding functional bugs and more about identifying usability issues. Feedback led to several UI improvements, such as clarifying icon tooltips, improving error message wording, and making the connection points on nodes larger and easier to click.

### 6.3 SAMPLE TEST CASES

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test Case ID** | **Feature** | **Test Description** | **Expected Result** | **Actual Result** | **Status** |
| TC-01 | User Login | Enter valid username and password. Click Login. | User is redirected to the builder page. The username appears in the header. | As expected | Pass |
| TC-02 | User Login | Enter invalid password. Click Login. | An error message "Invalid credentials" is displayed. User remains on the login page. | As expected | Pass |
| TC-03 | Canvas Interaction | Drag a CSVDataSource node and a StandardScaler node to the canvas. Connect the output of the first to the second. | A visual edge is drawn between the two nodes. The application state is updated to reflect the connection. | As expected | Pass |
| TC-04 | Code Generation | Build a simple pipeline: CSVDataSource -> LogisticRegression. Generate code. | A valid Python script is generated that imports pandas and scikit-learn, loads data, and defines the model. | As expected | Pass |
| TC-05 | Undo/Redo | Add a node, then delete it. Click Undo. | The deleted node reappears on the canvas in its original position. | As expected | Pass |
| TC-06 | Versioning | Save a pipeline. Create version "V1". Modify the pipeline. Load "V1". | The canvas reverts to the state it was in when "V1" was created. | As expected | Pass |
| TC-07 | Validation | Connect a LogisticRegression model's output to a StandardScaler's input. | The connection should be prevented, or a validation error should be displayed, as the data types are incompatible. | As expected | Pass |

### 6.4 TESTING ENVIRONMENT

All testing was conducted in an environment that closely mirrored the intended deployment setup.

1. **Operating Systems:** Windows 11, macOS Sonoma, and Ubuntu 22.04.
2. **Web Browsers:**
   * Google Chrome (Version 120+)
   * Mozilla Firefox (Version 118+)
   * Microsoft Edge (Version 120+)
3. **Backend Environment:**
   * Python 3.10
   * Flask development server (Werkzeug)
   * SQLite database file
4. **Code Execution Environment:** A separate virtual environment with pandas, scikit-learn, and other necessary libraries installed via pip to test the generated Python scripts.

## **CHAPTER 7: RESULTS AND SNAPSHOTS**

This chapter presents the tangible results of the project. The following screenshots showcase the final user interface of the DominoML application, illustrating the key features and workflows that were implemented. These visual results serve as a demonstration that the objectives outlined in Chapter 1 have been successfully met.

### 7.1 APPLICATION SCREENSHOTS

**Figure 7.1: User Authentication Page**The landing page provides a clean and modern interface for user signup and login. This is the entry point for accessing the application's core features.  
*(This screenshot would show the login/signup form from*

*(Placeholder for an actual screenshot of the login page)*

**Figure 7.2: The Main ML Pipeline Builder Interface**This is the central workspace of the application. The screenshot shows the three-panel layout: the Component Library on the left, the main Canvas in the center, and the context-aware Property Panel on the right. The dark theme is active, demonstrating the theme-toggling functionality.  
*(This screenshot would show the main builder view from*

*(Placeholder for an actual screenshot of the builder interface)*

**Figure 7.3: A Completed Visual Pipeline for a Classification Task**This image demonstrates a complete and logically connected pipeline. The color-coding for different component types (Data, Preprocessing, Model, Evaluation) is clearly visible, enhancing the readability of the workflow. A CSVDataSource is connected to a StandardScaler, which feeds into a LogisticRegression model.  
*(This screenshot would show a populated canvas with connected nodes)*

*(Placeholder for an actual screenshot of a built pipeline)*

**Figure 7.4: The Pipeline Versioning Management Modal**When a user clicks the "Versions" button in the toolbar, this modal appears. It displays the version history for the current pipeline, allowing the user to create new versions, add descriptions, and load previous snapshots.  
*(This screenshot would show the UI for the versioning system)*

![alt text](https.i.imgur.com/your-image-link-for-versions.png)

*(Placeholder for an actual screenshot of the versions modal)*

**Figure 7.5: Undo/Redo System in Action**The toolbar at the top contains the core action buttons, including Undo and Redo. Hovering over these buttons provides a tooltip describing the action to be undone or redone (e.g., "Undo: Add Node 'StandardScaler'"). This demonstrates the interactive history management system.  
*(This screenshot would be a close-up of the toolbar with the undo/redo buttons)*

*(Placeholder for a screenshot of the toolbar)*

**Figure 7.6: My Models Gallery**This view allows logged-in users to see all their saved pipelines. From here, they can choose to load a pipeline back into the builder for further editing or versioning.  
*(This screenshot would show the gallery of saved models)*

*(Placeholder for a screenshot of the model gallery)*

### 7.2 SAMPLE GENERATED CODE

The ultimate functional output of a pipeline is its executable Python code. Below is a sample script generated by DominoML for a basic classification pipeline. The code is intentionally structured to be clean, readable, and easy to understand, fulfilling one of the project's primary educational goals.

**Pipeline Description:** The visual pipeline consisted of loading a CSV file, selecting features and a target, scaling the features, and training a Logistic Regression model.

codePython

# Generated by DominoML - Flask Edition

# -------------------------

# 1. Import Libraries

# -------------------------

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score

print("--> [DominoML] Pipeline execution started.")

# -------------------------

# 2. Define Pipeline Steps

# -------------------------

# Step 1: Load Data from CSVDataSource

print("--> [Step 1] Loading data from CSVDataSource...")

try:

# Parameters from the 'CSVDataSource' node

file\_path = 'data/sample\_data.csv'

df = pd.read\_csv(file\_path)

print(f" Data loaded successfully. Shape: {df.shape}")

except FileNotFoundError:

print(f" [Error] File not found at path: {file\_path}")

exit()

# Step 2: Select Features and Target

print("--> [Step 2] Selecting features and target...")

# Parameters from the 'SelectFeatures' node

features = ['feature1', 'feature2', 'feature3']

target = 'target\_variable'

X = df[features]

y = df[target]

# Step 3: Split Data

print("--> [Step 3] Splitting data into training and testing sets...")

# Parameters from the 'TrainTestSplit' node

test\_size = 0.2

random\_state = 42

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=test\_size, random\_state=random\_state)

print(f" Training set size: {X\_train.shape[0]}, Testing set size: {X\_test.shape[0]}")

# Step 4: Scale Features with StandardScaler

print("--> [Step 4] Applying StandardScaler...")

# Parameters from the 'StandardScaler' node

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

# Step 5: Train LogisticRegression Model

print("--> [Step 5] Training LogisticRegression model...")

# Parameters from the 'LogisticRegression' node

C = 1.0

solver = 'lbfgs'

model = LogisticRegression(C=C, solver=solver, random\_state=random\_state)

model.fit(X\_train\_scaled, y\_train)

print(" Model training complete.")

# Step 6: Evaluate Model

print("--> [Step 6] Evaluating model performance...")

# Parameters from the 'ClassificationEvaluator' node

y\_pred = model.predict(X\_test\_scaled)

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"\n[RESULT] Model Accuracy: {accuracy:.4f}")

# -------------------------

# 3. Pipeline Execution Finished

# -------------------------

print("--> [DominoML] Pipeline execution finished.")

This generated script demonstrates the successful translation from a visual graph to a sequential, logical, and executable program.

## **CHAPTER 8: CONCLUSION AND FUTURE SCOPE**

This final chapter summarizes the project's achievements, provides an honest reflection on its limitations, and outlines potential directions for future development.

### 8.1 CONCLUSION

The project successfully delivered on its core objective: to create **DominoML**, a web-based, visual machine learning pipeline builder. The application effectively addresses the "abstraction gap" in modern ML practice by providing an intuitive, drag-and-drop interface that demystifies the structure of ML workflows. By abstracting away the boilerplate code, it lowers the barrier to entry for beginners and accelerates the prototyping process for experienced practitioners.

The key achievements of this project are:

1. **A Fully Functional Visual Builder:** The core drag-and-drop canvas, component library, and property panel were implemented, providing a seamless user experience.
2. **Robust Code Generation:** The system can reliably translate complex visual pipelines into clean, readable, and executable Python code, bridging the gap between low-code design and traditional development.
3. **Advanced Productivity Features:** The implementation of an undo/redo system with session persistence and a comprehensive pipeline versioning system elevates the tool from a simple builder to a practical platform for iterative model development and experiment tracking.
4. **A Secure, Modern Web Application:** Built on the Flask framework, the application incorporates user authentication, secure data persistence, and a responsive, modern UI with light and dark modes.

DominoML serves as a powerful educational tool and a rapid prototyping platform. It validates the hypothesis that visual workflow automation, a paradigm popularized by tools like n8n and Zapier, can be successfully applied to the domain of classical machine learning to enhance productivity and understanding.

### 8.2 LIMITATIONS OF THE CURRENT SYSTEM

An honest assessment requires acknowledging the project's limitations, which are primarily a result of deliberate scoping decisions to ensure a feasible and focused project.

1. **Limited Component Library:** The current library focuses exclusively on classical ML models within the Scikit-learn ecosystem. It does not include components for deep learning (TensorFlow, PyTorch), advanced statistical analysis, or data visualization.
2. **No Real-time Collaboration:** The system is designed for a single user at a time. It lacks features like Google Docs-style real-time collaboration on a single pipeline.
3. **Scalability for Big Data:** The code generation and data handling are designed for in-memory datasets using Pandas, making it unsuitable for big data workflows that would require technologies like Apache Spark.
4. **Basic Validation:** The real-time validation is primarily structural. It does not perform static analysis of the data itself (e.g., warning the user if they try to apply a StandardScaler to non-numeric data).

### 8.3 FUTURE SCOPE

The current version of DominoML provides a solid foundation. The project's modular architecture allows for numerous exciting avenues for future development. The GitHub README already outlines a clear roadmap for **Phase 3**, which represents the most immediate next step.

#### *8.3.1 Immediate Future Scope: Phase 3 - Export Runnable Artifacts*

As detailed in the project's documentation, the next logical evolution is to enhance the export capabilities significantly.

1. **Jupyter Notebooks:** Generate an interactive Jupyter Notebook (.ipynb) instead of just a Python script. This would be invaluable for educational purposes, as each step of the pipeline could be its own cell with explanatory Markdown text.
2. **Docker Containers:** Automatically generate a Dockerfile and a docker-compose.yml file. This would package the entire pipeline—including the Python script, a requirements.txt with pinned dependencies, and the dataset—into a self-contained, portable, and reproducible Docker container. This is a major step towards production-level MLOps.
3. **Command-Line Interface (CLI):** Enhance the generated Python script with argparse to turn it into a reusable CLI tool. This would allow users to run the pipeline from the command line and pass in arguments, such as the path to a new dataset.

#### *8.3.2 Long-Term Enhancements*

Beyond the immediate roadmap, other potential enhancements include:

1. **Expanded Component Library:** Gradually add new nodes for data visualization (e.g., Matplotlib or Plotly outputs), more advanced feature engineering techniques, and potentially wrappers for popular libraries like XGBoost and LightGBM.
2. **Live Model Deployment:** Integrate with cloud services (like AWS Lambda or Google Cloud Functions) to add a "Deploy as API" button, which would automatically package and deploy a trained model as a REST API endpoint.
3. **Enhanced UI/UX:** Implement features like multi-node selection, copy-pasting nodes, and adding comments/notes directly onto the canvas to improve the user experience for complex pipelines.
4. **Git Integration:** Allow users to sync their saved models and versions with a GitHub repository, providing a more robust and familiar version control experience for developers.

This project has laid the groundwork for a versatile and user-friendly ML platform. With these future enhancements, DominoML has the potential to grow into an even more powerful tool for the data science community.

## **REFERENCES**

M. G. (2023). *Flask Web Development, 2nd Edition*. O'Reilly Media.  
*(A foundational text for understanding the Flask framework used in the backend.)*

A. Géron (2022). *Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow, 3rd Edition*. O'Reilly Media.  
*(Provided core concepts and best practices for implementing Scikit-learn pipelines, which heavily influenced the code generation logic.)*

SQLAlchemy Documentation. (2023). *SQLAlchemy 2.0 ORM*. Retrieved from <https://docs.sqlalchemy.org/en/20/orm/>*(The official documentation was the primary resource for implementing the database models and interactions.)*

Flask-Login Documentation. (2023). *Flask-Login 0.6.2*. Retrieved from <https://flask-login.readthedocs.io/en/latest/>*(Essential for implementing secure user session management.)*

n8n.io. (2023). *Workflow Automation Tool*. Retrieved from <https://n8n.io/>*(Inspiration for the visual, node-based workflow automation paradigm.)*

KNIME AG. (2023). *KNIME Analytics Platform*. Retrieved from <https://www.knime.com/>*(Reviewed as a key example of an existing desktop-based visual ML tool.)*

Mozilla Developer Network (MDN). (2023). *JavaScript Web APIs*. Retrieved from <https://developer.mozilla.org/en-US/docs/Web/API>*(The primary reference for all vanilla JavaScript implementation, including DOM manipulation, event handling, and Fetch API for server communication.)*

L. S. et al. (2023). *Lucide Icons*. Retrieved from <https://lucide.dev/>*(The open-source icon library used for the application's user interface.)*

## **APPENDIX A: CORE PYTHON CODE SNIPPETS**

This appendix contains selected code snippets from the backend to highlight key implementation patterns.

**A.1: Flask Application Factory (**

codePython

from flask import Flask

from flask\_sqlalchemy import SQLAlchemy

from flask\_login import LoginManager

from config import Config

db = SQLAlchemy()

login\_manager = LoginManager()

login\_manager.login\_view = 'auth.login'

def create\_app(config\_class=Config):

app = Flask(\_\_name\_\_)

app.config.from\_object(config\_class)

db.init\_app(app)

login\_manager.init\_app(app)

from app.models import User

@login\_manager.user\_loader

def load\_user(user\_id):

return User.query.get(int(user\_id))

# Register blueprints

from app.routes.main import bp as main\_bp

app.register\_blueprint(main\_bp)

from app.routes.auth import bp as auth\_bp

app.register\_blueprint(auth\_bp, url\_prefix='/auth')

from app.routes.api import bp as api\_bp

app.register\_blueprint(api\_bp, url\_prefix='/api')

with app.app\_context():

db.create\_all() # Ensure DB is created

return app

**A.2: API Endpoint for Saving a Model (**

codePython

from flask import request, jsonify

from flask\_login import login\_required, current\_user

from app import db

from app.models import SavedModel

# ... (inside the api blueprint) ...

@bp.route('/models/save', methods=['POST'])

@login\_required

def save\_model():

data = request.get\_json()

if not data:

return jsonify({'error': 'Invalid request body'}), 400

model\_id = data.get('model\_id')

name = data.get('name')

graph\_json = data.get('graph\_json')

if model\_id:

# Update existing model

model = SavedModel.query.get(model\_id)

if model.user\_id != current\_user.id:

return jsonify({'error': 'Unauthorized'}), 403

model.name = name

model.graph\_json = graph\_json

else:

# Create new model

model = SavedModel(

name=name,

graph\_json=graph\_json,

user\_id=current\_user.id

)

db.session.add(model)

db.session.commit()

return jsonify({'message': 'Model saved successfully', 'model\_id': model.id}), 200

## **APPENDIX B: CORE JAVASCRIPT CODE SNIPPETS**

This appendix shows key snippets from the vanilla JavaScript frontend that power the interactive canvas.

**B.1: Main Rendering Loop Concept (**

codeJavaScript

// Global state object

let canvasState = {

nodes: [],

edges: [],

// ... other properties like panning offset

};

function renderCanvas() {

// Clear the entire canvas

context.clearRect(0, 0, canvas.width, canvas.height);

// 1. Render all edges (connections) first, so they are behind nodes

canvasState.edges.forEach(edge => {

const startNode = canvasState.nodes.find(n => n.id === edge.from.nodeId);

const endNode = canvasState.nodes.find(n => n.id === edge.to.nodeId);

// ... logic to calculate start and end points and draw a line

drawLine(startNode.x, startNode.y, endNode.x, endNode.y);

});

// 2. Render all nodes on top of the edges

canvasState.nodes.forEach(node => {

// ... logic to draw the node's rectangle, text, and connection points

drawNode(node);

});

}

// Any function that modifies the state calls renderCanvas() at the end

function addNode(nodeData) {

canvasState.nodes.push(nodeData);

// After changing state, re-render everything

renderCanvas();

}

**B.2: History Management for Undo/Redo (**

codeJavaScript

const MAX\_HISTORY\_SIZE = 50;

let undoStack = [];

let redoStack = [];

function saveState(currentState) {

// A real implementation requires a deep copy of the state object

const stateCopy = JSON.parse(JSON.stringify(currentState));

undoStack.push(stateCopy);

// Clear the redo stack whenever a new action is taken

redoStack = [];

if (undoStack.length > MAX\_HISTORY\_SIZE) {

undoStack.shift(); // Keep the stack size manageable

}

updateToolbarButtons(); // Enable/disable undo/redo buttons

}

function undo() {

if (undoStack.length > 1) { // Keep at least one initial state

const currentState = undoStack.pop();

redoStack.push(currentState);

const previousState = undoStack[undoStack.length - 1];

// Load the previous state onto the canvas (deep copy)

loadCanvasState(JSON.parse(JSON.stringify(previousState)));

}

updateToolbarButtons();

}

// The redo function works in the opposite direction

## **APPENDIX C: ENVIRONMENT SETUP AND INSTALLATION GUIDE**

This guide provides the steps to set up and run the DominoML application on a local machine.

**Prerequisites**

Python 3.8 or higher

pip (Python package manager)

Git

**Clone the Repository**Open a terminal or command prompt and clone the project from GitHub:

codeBash

git clone https://github.com/1mystic/Domino\_ML.git

cd Domino\_ML

**Create and Activate a Virtual Environment**It is highly recommended to use a virtual environment to manage dependencies.

**On Windows:**

codeBash

python -m venv venv

venv\Scripts\activate

**On macOS/Linux:**

codeBash

python3 -m venv venv

source venv/bin/activate

**Install Dependencies**Install all required Python packages using the requirements.txt file:

codeBash

pip install -r requirements.txt

**Configure Environment Variables**The application uses environment variables for configuration. Create a .env file in the root directory of the project by copying the example file:

codeBash

# On Windows:

copy .env.example .env

# On macOS/Linux:

cp .env.example .env

Open the newly created .env file and ensure the following variables are set. The default values are suitable for local development. You should change the SECRET\_KEY for any serious deployment.

codeCode

SECRET\_KEY=your-secret-key-here-change-this

DATABASE\_URL=sqlite:///glideml.db

FLASK\_ENV=development

FLASK\_DEBUG=True

**Initialize the Database**The first time you run the application, the database needs to be created. The create\_all() command in the application factory handles this automatically.

**Run the Application**Start the Flask development server with the following command:

codeBash

python run.py

You should see output indicating that the server is running, typically on http://127.0.0.1:5000 or http://localhost:5000.

**Access the Application**Open a modern web browser and navigate to http://localhost:5000. You should see the DominoML landing page. You can now register a new account and start building ML pipelines.