SnapPy: Technical Guide

Authors: Luigi Di Paolo, Andy Vodopi
Supervisor: Dr Stephen Blott

Contents

Abstract 3

1 Introduction 3
- 1.1 Glossary 3
- 1.2 Research and Motivation 5
  - 1.2.1 The Challenge of Learning Programming 5
  - 1.2.2 Block-Based Programming as a Solution 5
  - 1.2.3 Python as an Ideal Learning Language 5
  - 1.2.4 Existing Platforms 5

2 Design 6
- 2.1 High-Level Architecture 6
- 2.2 Architecture Diagram 7
- 2.3 Frontend Architecture 7
  - 2.3.1 UI Design 8
- 2.4 Backend Architecture 8
- 2.5 JWT Authentication 9
- 2.6 Load Project 10
- 2.7 Edit Canvas 11

3 Implementation 11
- 3.1 Block System 11
  - 3.1.1 Block Type Hierarchy 11
  - 3.1.2 Block Relationships 12
  - 3.1.3 Block States and Shapes 13
  - 3.1.4 Specialised Block Types 13
  - 3.1.5 Drop Zones and Connection Points 14
  - 3.1.6 withBlock HOC (Higher-Order Component) 15
  - 3.1.7 Blocks Rendering 16
- 3.2 State Management 18
  - 3.2.1 Canvas State Management 19
  - 3.2.2 Block Actions 19
  - 3.2.3 Blocks Reducer 20
  - 3.2.4 Blocks Context 22
- 3.3 Code Generation 23
  - 3.3.1 Code Generation Architecture 23
  - 3.3.2 Indentation Management 24

3.3.3 Block-Specific Code Generation . . . 24
3.3.4 Expression Handling . . . 25
3.4 Python Interpretation . . . 25
3.4.1 Pyodide Integration . . . 26
3.4.2 Custom Hook for Code Execution . . . 26
3.4.3 Code Execution Process . . . 28
3.4.4 Error Handling . . . 29
3.5 Backend Implementation . . . 29
3.5.1 Database Model . . . 29
3.5.2 Entity Relationship Diagram . . . 30
3.5.3 API Design . . . 31

4 Problems Solved **32**
4.1 Block Rendering Approach . . . 32
4.2 Canvas Data Structure Evolution . . . 33
4.3 Drag and Drop Implementation . . . 33
4.4 Minimising Runtime Errors . . . 33
4.5 Unfamiliar Technical Territory . . . 34
4.6 Finding an Appropriate Workflow . . . 34

5 Results **36**
5.1 Testing Strategy . . . 36
5.2 Testing Tools . . . 36
5.3 Block System Tests . . . 37
5.3.1 Block Manipulation Tests . . . 37
5.3.2 Block Connection Tests . . . 37
5.3.3 Nested Block Tests . . . 38
5.4 Utility Function Tests . . . 39
5.5 Code Generation Tests . . . 40
5.6 UI Component Tests . . . 41
5.7 Test Coverage . . . 43

6 Future Work **43**
6.1 Enhanced Block Types . . . 44
6.2 Advanced Debugging Tools . . . 44
6.3 Real-Time Collaboration . . . 44
6.4 Mobile Support . . . 44

References **45**

Abstract

SnapPy is a novel block-based visual programming environment designed to make learning Python more accessible and engaging for beginners. It bridges the gap between visual block-based programming and text-based programming by allowing users to create Python code by manipulating drag-and-drop blocks. As users build programs visually, SnapPy simultaneously generates the equivalent Python code, enabling a smooth transition from block-based thinking to text-based coding. The system features workspace management, real-time code generation and an integrated Python interpreter, all within a modern single page app[1] built using Next.js[2].

1 Introduction

The ability to code is increasingly becoming a fundamental literacy in our technology-driven world. However, learning to program remains challenging for many beginners, who often struggle with syntax errors and the abstract nature of programming concepts. Snap-Py addresses this challenge by creating a bridge between visual block-based programming environments and traditional text-based coding in Python. Snap-Py provides a scaffolded learning experience where users can create Python programs using intuitive drag-and-drop blocks, see the equivalent Python code generated in real-time and execute their programs in an integrated interpreter.
This technical guide provides an in-depth look at the design, architecture, and implementation of Snap-Py, including the challenges faced during development and the solutions applied.

1.1 Glossary

Term	Definition
AST (Abstract Syntax Tree)	A data structure used to represent the structure of a program.
Block	The fundamental visual element in SnapPy representing a Python programming construct.
Canvas	The workspace area where users arrange blocks to create programs.
Component	A reusable piece of UI in React that encapsulates structure, behavior, and style.
dnd-kit	The drag-and-drop library used for block manipulation.

Term	Definition
Extreme Programming	An agile software development methodology focusing on frequent releases, continuous testing, and pair programming to improve software quality.
GitLab	A DevOps platform providing source code management, CI/CD pipelines, issue tracking, and collaborative tools for software development teams.
JavaScript	An interpreted programming language mostly used for web pages.
Kanban Board	A visual workflow management tool that helps teams visualize work, limit work-in-progress, and maximize efficiency through columns representing different stages of work.
Higher-Order Component (HOC)	A pattern used in React to wrap components with common functionality.
Immutability	The practice of not modifying state directly but creating new state objects.
Mantine UI	A modern React component library.
Next.js	The React-based framework used to build the SnapPy application.
ORM (Object-Relational Mapping)	A programming technique for converting data between incompatible systems in object-oriented programming.
Prisma	The Object-Relational Mapping (ORM) tool used for database operations.
Pyodide	WebAssembly-based Python interpreter enabling client-side Python execution.
React	A JavaScript library for building user interfaces with a component-based architecture.
React Context	System used to make state and actions available throughout the component tree.
Reducer Pattern	State management approach that centralizes state changes through well-defined actions.
Type Safety	Property of a program that involves the use of a type system to help prevent errors at compile time.
TypeScript	A strongly typed programming language that builds on JavaScript.
WebAssembly	A binary instruction format that enables high-performance execution of code in web browsers.
Workbench	Area containing template blocks that users can drag onto the canvas.

1.2 Research and Motivation

1.2.1 The Challenge of Learning Programming

Research by Weeda [3] et al. highlights that beginners often struggle with programming due to the high cognitive load associated with learning complex syntax, difficulty grasping abstract programming concepts, lack of immediate feedback on errors or logical gaps, and frustration when facing syntax errors that prevent code execution. These issues can create significant barriers for novice programmers, leading many to become discouraged before they develop a solid foundation.

1.2.2 Block-Based Programming as a Solution

Studies by Weintrop & Wilensky [4] demonstrate that the visual approach of block-based programming significantly improves algorithmic thinking, allows users to focus on core concepts like loops, conditionals, and data flow without being overwhelmed by syntax, engages users through a more intuitive and enjoyable interface, and reduces the initial learning curve for programming.

1.2.3 Python as an Ideal Learning Language

Python was chosen as the target language for SnapPy because of its suitability for beginners. According to Cutting & Stephen [5], Python is an ideal language for novices due to its readable and concise syntax and high level of abstraction. The language minimises distractions from complex syntax rules, allowing learners to concentrate on fundamental programming concepts. Python's versatility across numerous fields including data science, web development, automation, and scientific computing ensures that skills learned through SnapPy remain relevant across diverse career paths.

1.2.4 Existing Platforms

We conducted an analysis of established block-based programming platforms, examining their pedagogical approaches, technical implementations, and transition pathways to text-based languages. Despite the educational value these platforms provide, none fully addressed our specific objective of creating a seamless bridge between visual programming and Python literacy.

1.2.4.1 Scratch Scratch[6] offers an excellent introduction to programming concepts through its intuitive block-based interface. However, it works within its own isolated ecosystem with a proprietary block language that doesn't translate directly to regular programming languages. While Scratch excels at teaching fundamental programming logic, its structure differ significantly from Python. SnapPy specifically bridges this gap by maintaining the visual accessibility of blocks while directly teaching Python's conventions and structure.

1.2.4.2 Microbit Microbit's block editor provides a simplified programming environment targeted specifically at physical computing applications[7]. While it does generate Python code behind the scenes, its block language is constrained to the capabilities of the Microbit hardware, limiting the range of programming concepts that can be explored. SnapPy, in contrast, aims to teach general-purpose Python programming applicable across diverse domains.

2 Design

2.1 High-Level Architecture

As with most web applications, SnapPy is divided into two main components: frontend and backend. The backend is deliberately minimalist, handling only authentication and database interactions. This architectural decision places the vast majority of the computational complexity, and consequently most of the codebase, in the frontend. We made this design choice to ensure nearly all operations occur client-side, eliminating latency from server roundtrips and providing users with a responsive, seamless experience regardless of network conditions.

2.2 Architecture Diagram

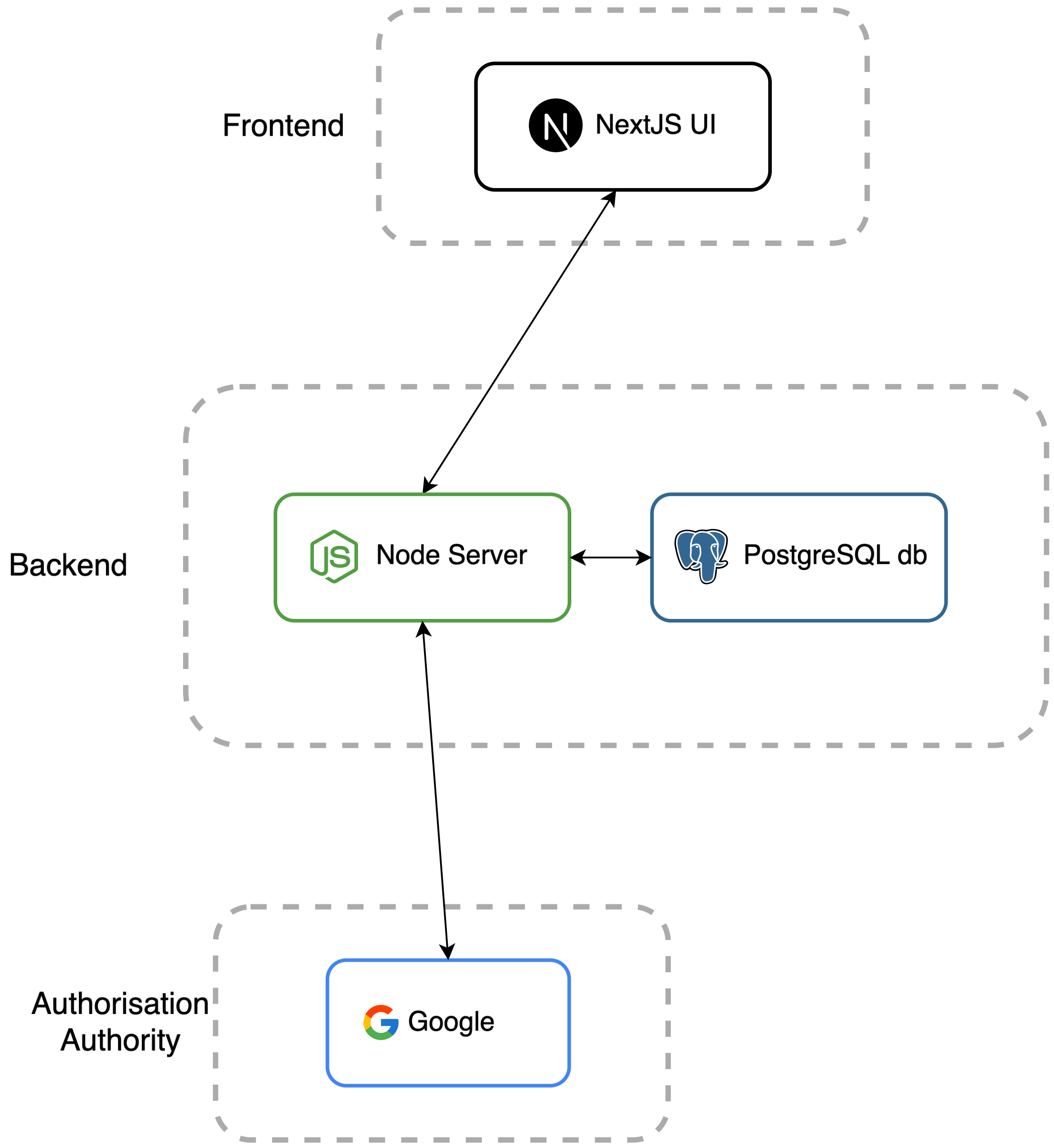


Figure 1: Architecture Diagram

2.3 Frontend Architecture

The frontend is built using Next.js, a React-based framework chosen for its server-side rendering capabilities, static site generation options, built-in performance optimizations, and seamless integration with React components.

Key frontend components include:

- User interface components (Canvas, Workbench, Code Editor, Output Panel)
- Block rendering system
- Drag-and-drop functionality (using dnd Kit [8])
- Code generation
- Code editor with syntax highlighting
- Python interpreter integration

2.3.1 UI Design

The user interface employs Mantine UI as a main component library, with tailored customizations to maintain a cohesive look throughout the application. This approach balanced development efficiency with the need for a cohesive and modern-looking feeling system.
For the block system, we implemented a purposeful visual design strategy focusing on user comprehension. The blocks have distinctive shapes and connection points that visually communicate their functionality and relationship possibilities. Square blocks represent statements that perform actions, while round blocks indicate expressions that return values. This intuitive shape-based grammar significantly reduces the cognitive load for novice programmers while reinforcing programming concepts through visual patterns.

2.4 Backend Architecture

The backend uses Node.js with a PostgreSQL database to provide:

- User authentication and session management
- Data persistence for user projects

We used Prisma [9] as our ORM to streamline database operations and to ensure type safety between our backend services and the database schema.

2.5 JWT Authentication

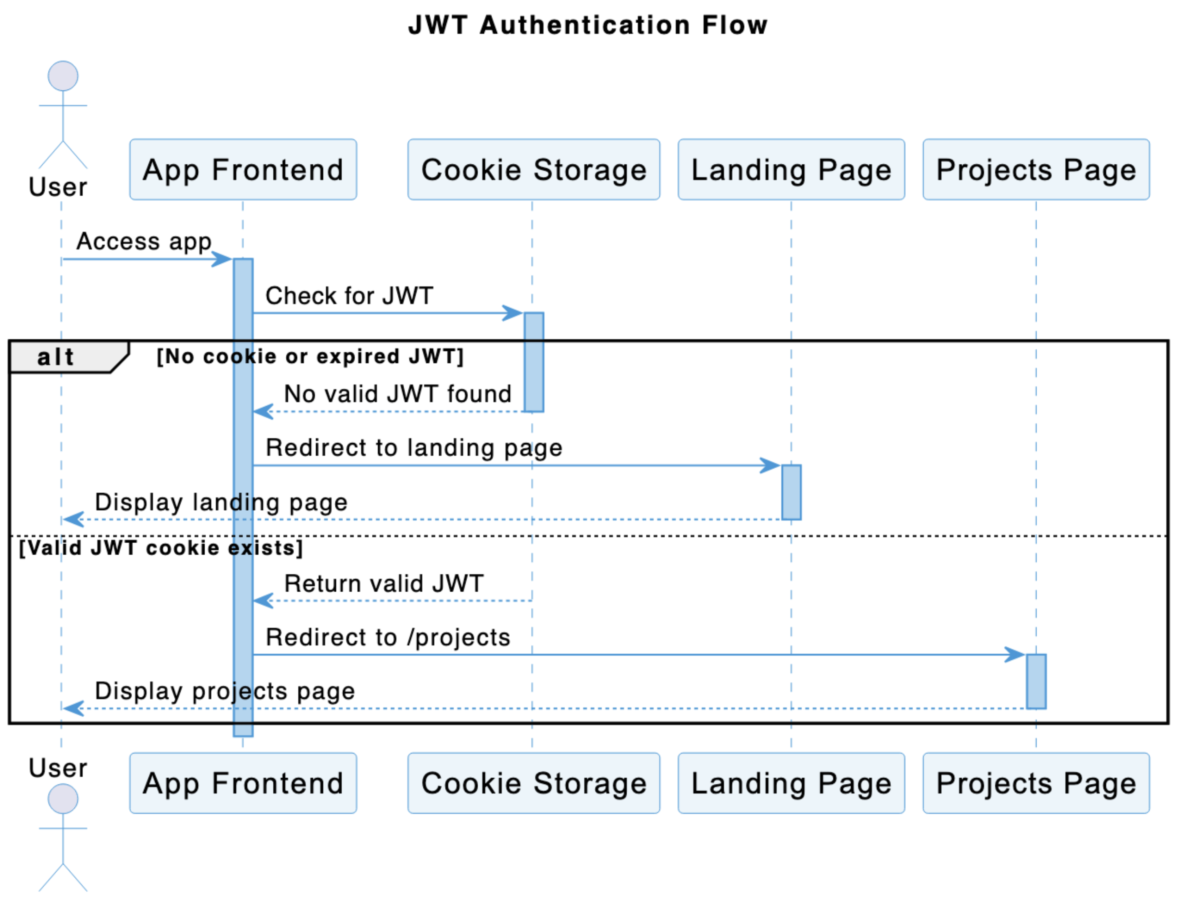


Figure 2: Authentication Flow

2.6 Load Project

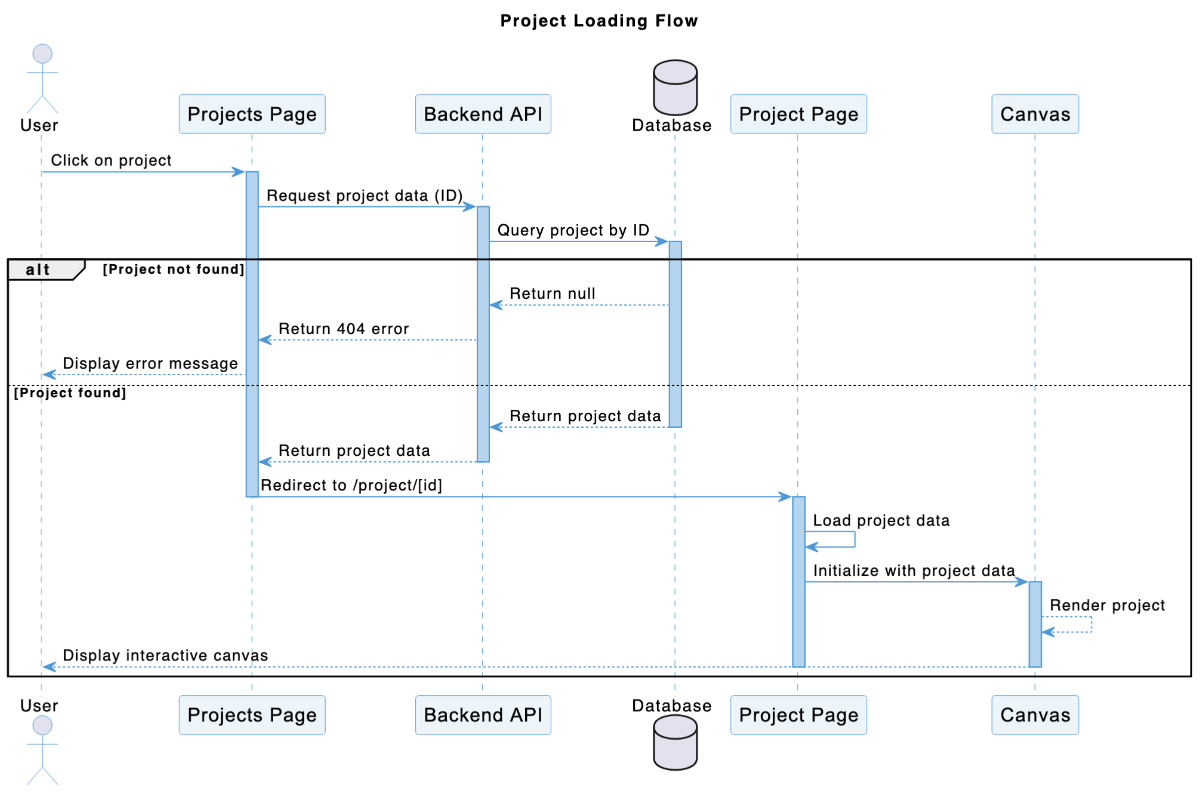


Figure 3: Load Project

2.7 Edit Canvas

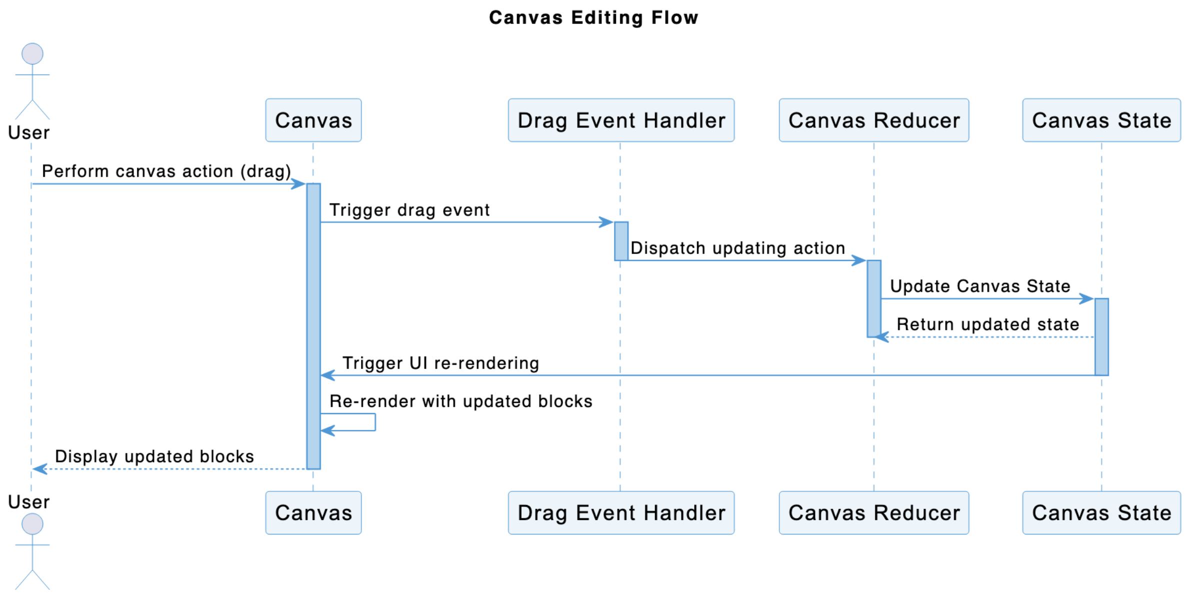


Figure 4: Edit Canvas

3 Implementation

3.1 Block System

The block system is the core of SnapPy and is designed to represent Python programming constructs visually. Each block corresponds to a specific Python programming element, carefully crafted to maintain fidelity with the underlying language while providing an intuitive visual representation that abstracts away syntax complexities.

3.1.1 Block Type Hierarchy

Blocks are defined using TypeScript interfaces with a comprehensive type system:

```
interface BaseBlock {
  id: string;
  type: BlockType;
  shape: BlockShape;
  state: BlockState;
  coords: Coordinates;
  isWorkbenchBlock: boolean;
  parentId: string | null;
  prevId: string | null;
  nextId: string | null;
}
```

This base interface is meant to be then extended by every block to adapt to their custom behaviours. For example a While Loop block will extend it like so:

```
export interface WhileBlock extends BaseBlock {
  type: BlockType.While;
  shape: BlockShape.Square;
  children: {
    condition: Block[];
    body: Block[];
  };
}
```

All blocks are then exported under a single Union called Block:

```
export type Block =
  | ProgramStartBlock
  | VariableBlock
  | VariableValueBlock
  | WhileBlock
  | NumberBlock
  | StringBlock
  | MathBlock
  | EmptyBlock
  | BooleanBlock
  | ComparisonBlock
  | LogicalBlock
  | IfBlock
  | IfElseBlock
  | ForBlock
  | PrintBlock;
```

This robust type system ensures type safety throughout the application, enables specialised block behaviors based on type, maintains consistent structure across all blocks, and facilitates straightforward extension with new block types as the system evolves.

3.1.2 Block Relationships

The block system implements two types of relationships:

1. Sequential relationships: Blocks that follow each other in a sequence (vertical stacking) are managed through `prevId` and `nextId` properties. This forms a doubly-linked list structure for

efficient traversal and allows blocks to be connected in a linear flow representing sequential execution.

2. Hierarchical relationships: Blocks nested inside other blocks are managed through `parentId` and `children` properties. This architecture supports complex nested structures like loops and conditionals while enabling proper expression evaluation within specific block contexts.

This dual relationship system allows SnapPy to accurately represent the full range of Python programming constructs visually while maintaining the logical structure necessary for proper code generation and execution.

3.1.3 Block States and Shapes

Blocks have different states and shapes that affect their appearance and behaviour:

```
export enum BlockState {
  Idle = "idle",
  Selected = "selected",
  Dragging = "dragging",
  Nested = "nested",
}

export enum BlockShape {
  Round = "round",
  Square = "square",
}
```

States control visual feedback and interaction behaviors within the interface. The Idle state represents the default block appearance, while Selected provides visual indication of user focus. The Dragging state activates during user manipulation, and Nested applies specific styling when blocks are contained within other blocks. Shapes serve as visual indicators of block compatibility and function. Round blocks represent expressions that return values (like variables or mathematical operations), while Square blocks indicate statements that perform actions (such as control flow or print operations). This shape-based visual language helps users intuitively understand which blocks can connect together.

3.1.4 Specialised Block Types

Different block types have specialised structures to represent their functionality:

```
// Example: Variable assignment block
export interface VariableBlock extends BaseBlock {
  type: BlockType.Variable;
  shape: BlockShape.Square;
  selected: string;
  children: {
    expression: Block[];
  };
}

// Example: Math operation block
export interface MathBlock extends BaseBlock {
  type: BlockType.Math;
  shape: BlockShape.Round;
  operator: MathOperator;
  children: {
    left: Block[];
    right: Block[];
  };
}

// Example: While loop block
export interface WhileBlock extends BaseBlock {
  type: BlockType.While;
  shape: BlockShape.Square;
  children: {
    condition: Block[];
    body: Block[];
  };
}
```

These examples represent just a small subset of the approximately 17 specialised block types in the system. Each block type extends the base interface with properties that capture its unique functionality. This comprehensive collection covers a range of Python programming constructs needed for effective learning, from expressions control structures, all while maintaining a consistent interface throughout the system.

3.1.5 Drop Zones and Connection Points

To facilitate block connections, the system implements various types of drop zones:

```
export enum OuterDropzonePosition {
  Top = "top",
  Bottom = "bottom",
}

export enum InnerDropzoneShape {
  Round = "round",
  Square = "square",
}
```

Outer Drop Zones allow blocks to be connected in sequence. The Top position enables connecting

a block before the current block, while the Bottom position facilitates connecting a block after the current block. This creates the vertical stacking behavior essential for sequential code execution.
Inner Drop Zones enable blocks to be nested within container blocks. Expression slots accept round blocks that form conditions or expressions, while Statement slots accommodate square blocks that create control flow structures.
The system visually highlights these drop zones during drag operations, providing immediate feedback on valid connection points and helping users understand the structural relationships between different code elements.

3.1.6 withBlock HOC (Higher-Order Component)

All blocks are wrapped by a Higher-Order Component. A HOC is simply a React component that wraps another component to add some functionality:

```
// Higher-Order Component for common block functionality
export default function withBlock<T extends object>(
  WrappedBlock: React.ComponentType<T>
) {
  const WithBlock = (props: T & WithBlockProps) => {
    // Common block rendering logic
    return (
      <div ref={setNodeRef} {...attributes} {...listeners}>
        {/* Block UI with drop zones */}
        <div className={classes.base} style={blockStyles}>
          <WrappedBlock
            id={id}
            isWorkbenchBlock={isWorkbenchBlock}
            {...restProps}
          >
            {children}
          </WrappedBlock>
        </div>
        {/* Drop zones for block connections */}
      </div>
    );
  };
  return WithBlock;
}
```

This pattern effectively separates shared functionality from block-specific logic, resulting in remarkably clean code for individual blocks. The HOC handles all common block behaviors including drag-and-drop interactions, drop zone rendering, and styling, while the wrapped components focus solely on their unique visual representation and behavior.

```
import withBlock from '../components/with-block';

```

```
function Empty() {
  return <span>empty Block</span>;
}

export default withBlock(Empty);
```

The code above demonstrates the Empty Block implementation, a simple block used for testing. Despite its minimal appearance, the block inherits all necessary features to function properly within the canvas environment, including drag-and-drop capabilities, connection points, and state management.

3.1.7 Blocks Rendering

Blocks are rendered by a Renderer component that systematically converts the block data structure into visual elements:

```
export default function CanvasBlocksRenderer({
  canvas,
  enableSequences,
}: CanvasBlocksRendererProps) {
  // Find all start blocks
  const startBlocks = canvas.filter((block) => block.prevId === null);

  return (
    <>
      {startBlocks.map((block) => {
        return (
          // Find sequence of blocks starting from start block and
             render it
          <BlockGroupWrapper
            key={block.id}
            sequence={getBlocksSequence(block, canvas)}
            enableSequences={enableSequences}
          />
        );
      })}
    </>
  );
}
```

This component finds the entry points of block sequences by locating blocks with no predecessors. It then processes each sequence through the BlockGroupWrapper component, which provides drag-and-drop functionality for the entire block group:

```
// Wraps all blocks in a sequence with a draggable div
function BlockGroupWrapper({
  sequence,
  enableSequences,
}: BlockGroupWrapperProps) {
  const startBlock = sequence[0];
  const { id, coords } = startBlock;
  const { state } = useBlocks();

  const { attributes, listeners, setNodeRef, transform } = useDraggable
     ({
    id: id,
  });

  const style: React.CSSProperties = {
    transform: CSS.Translate.toString(transform),
    position: startBlock.parentId ? 'static' : 'absolute', // Use
       position static if nested, otherwise position absolutely
    top: coords.y,
    left: coords.x,
    zIndex: state.draggedGroupBlockIds?.has(id) ? 10 : 1,
  };

  return (
    <div
      key={`sequence_${id}`} // id of a sequence is always 'sequence_'
         + start block id
      ref={setNodeRef}
      style={style}
      {...listeners}
      {...attributes}
    >
      {sequence.map((block) => renderBlock(block, enableSequences))}
    </div>
  );
}
```

The renderBlock function is essentially a factory that renders the appropriate component based on block type:

```
export default function renderBlock(block: Block, enableSequences:
    boolean) {
  const { id, prevId, isWorkbenchBlock, state, shape, children } =
      block;

  switch (block.type) {
    case BlockType.ProgramStart:
      return (
        <ProgramStart
          key={block.id}
          id={id}
          hasPrev={prevId !== null}
          isWorkbenchBlock={isWorkbenchBlock}
          blockType={BlockType.ProgramStart}
          blockShape={shape}
```

```
            blockState={state}
            enableSequences={enableSequences}
          >
            {children}
          </ProgramStart>
        );
      case BlockType.Variable:
        return (
          <Variable
            key={id}
            id={id}
            hasPrev={prevId !== null}
            isWorkbenchBlock={isWorkbenchBlock}
            blockType={BlockType.Variable}
            blockState={state}
            blockShape={shape}
            selected={block.selected}
            enableSequences={enableSequences}
          >
            {block.children}
          </Variable>
        );
         // more cases...
      default:
        return (
          <Empty
            key={block.id}
            id={id}
            hasPrev={prevId !== null}
            isWorkbenchBlock={isWorkbenchBlock}
            blockType={BlockType.Empty}
            blockShape={shape}
            blockState={state}
            enableSequences={enableSequences}
          >
            {children}
          </Empty>
        );
    }
}
```

3.2 State Management

SnapPy implements a sophisticated state management system using the Reducer pattern [10] to handle the complex state requirements of a visual programming environment.

3.2.1 Canvas State Management

The state of the entire canvas is managed through a central state object:

```
export interface CanvasState {
  workbench: Block[];
  canvas: Block[];
  variables: string[];
  selectedBlockId: string | null;
  draggedBlockId: string | null;
  draggedGroupBlockIds: Set<string> | null;
  highlightedDropZoneId: string | null;
  entrypointBlockId: string | null;
}
```

This state structure encapsulates all dynamic elements of the programming environment. The workbench array contains template blocks available for use, while the canvas array holds the actual program being constructed. The variables collection tracks all user-defined variables for reference throughout the system. The remaining properties manage interaction states, tracking which block is currently selected, which block or group is being dragged, which drop zone is highlighted for potential connections, and which block serves as the program's entry point for execution.

3.2.2 Block Actions

User interactions are translated into a set of well-defined actions:

```
export enum CanvasEvent {
  SELECT_BLOCK = "select block",
  DESELECT_BLOCK = "deselect block",
  START_DRAG = "start drag",
  END_DRAG = "end drag",
  CREATE_BLOCK = "create block",
  DELETE_BLOCK = "delete block",
  CREATE_VARIABLE = "create variable",
  CHANGE_VARIABLE_SELECTED_OPTION = "change variable selected option",
  CHANGE_VARIABLE_VALUE_SELECTED_OPTION = "change variable value
      selected option",
  ADD_CHILD_BLOCK = "add child block",
  REMOVE_CHILD_BLOCK = "remove child block",
  SNAP_BLOCK = "snap block",
  UNSNAP_BLOCK = "unsnap block",
  UPDATE_BLOCK = "update block",
  HIGHLIGHT_DROPZONE = "highlight dropzone",
  CLEAR_HIGHLIGHTED_DROPZONE = "clear highlighted dropzone",
  CHANGE_INPUT_TEXT = "change input text",
  CHANGE_BOOLEAN_VALUE = "change boolean value",
}
```

Each action is represented by an interface with appropriate type and payload:

```
interface SelectBlock {
  type: CanvasEvent.SELECT_BLOCK;
  payload: { id: string };
}

interface DeselectBlock {
  type: CanvasEvent.DESELECT_BLOCK;
}

// Other action types...

export type CanvasAction =
    | SelectBlock
    | DeselectBlock
    | StartDrag
    | EndDrag
    | CreateBlock
    | DeleteBlock
    | CreateVariable
    | ChangeVariableSelectedOption
    | ChangeVariableValueSelectedOption
    | AddChildBlock
    | RemoveChildBlock
    | SnapBlock
    | UnsnapBlock
    | UpdateBlock
    | HighlightDropzone
    | ClearHighlightedDropzone
    | HighlightDropzone
    | ClearHighlightedDropzone
    | ChangeInputText
    | ChangeBooleanValue;
```

This action system maps all possible user interactions within the environment. Each interaction triggers a specific, strongly-typed action that carries precisely the data needed to process the state change. This approach ensures predictable state transitions.

3.2.3 Blocks Reducer

The core of the state management system is the `BlocksReducer` function, which handles all state transitions:

```
export default function BlocksReducer(
  state: CanvasState,
  action: CanvasAction,
) {
  switch (action.type) {
    case CanvasEvent.SELECT_BLOCK: {
      const { id } = action.payload;
      const block = validateBlockExists(
        state.canvas,
        id,
        CanvasEvent.SELECT_BLOCK,
      );
      if (!block) return state;

      const newBlock = { ...block, state: BlockState.Selected };
      return {
        ...state,
        canvas: updateBlockById(state.canvas, id, newBlock),
        selectedBlockId: id,
      };
    }

    case CanvasEvent.DESELECT_BLOCK: {
      const id = state.selectedBlockId;
      if (!id) return state;

      const block = validateBlockExists(
        state.canvas,
        id,
        CanvasEvent.DESELECT_BLOCK,
      );
      if (!block) return state;

      const newBlock = { ...block, state: BlockState.Idle };
      return {
        ...state,
        canvas: updateBlockById(state.canvas, id, newBlock),
        selectedBlockId: null,
      };
    }

    // Other action handlers...

    default:
      return state;
  }
}
```

This reducer follows immutability principles to simplify state management [11]. It takes the current state and an action, validates the action parameters to ensure operation safety, creates a modified state that reflects the action's intended effect, and returns a new state object that triggers appropriate UI updates. Each case in the switch statement handles a specific action type, applying focused changes to only the relevant parts of the state while preserving other state values.

3.2.4 Blocks Context

The state and actions are made available to components through a React Context:

```
export const BlocksContext = createContext<BlocksContextType | null>(
    null);

// Custom hook for accessing the context
export const useBlocks = () => {
  const context = useContext(BlocksContext);
  if (!context) {
    throw new Error('useBlocks must be used within a BlocksProvider');
  }
  return context;
};

export default function BlocksProvider({
  children,
  canvas,
  variables,
}: BlocksProviderProps) {
  const startBlock = findBlockById('start', canvas)!;
  const initialState: CanvasState = {
    workbench: workbenchBlocks,
    canvas: canvas,
    variables,
    selectedBlockId: null,
    draggedBlockId: null,
    draggedGroupBlockIds: null,
    highlightedDropZoneId: null,
    entrypointBlockId: startBlock.nextId,
  };

  const [state, dispatch] = useReducer(BlocksReducer, initialState);

  // Action creator functions
  const selectBlockAction = (id: string) => {
    dispatch({
      type: CanvasEvent.SELECT_BLOCK,
      payload: { id },
    });
  };

  const deselectBlockAction = () => {
    dispatch({
      type: CanvasEvent.DESELECT_BLOCK,
    });
  };

  // Other action creators...

  const value: BlocksContextType = {
    state,
    selectBlockAction,
    deselectBlockAction,
    // Other actions...
  };

  return (
    <BlocksContext.Provider value={value}>{children}</BlocksContext.
        Provider>
  );
}
```

This context provider establishes the state management backbone of the application. It initialises the state with values fetched from the database, creates a reducer linked to this initial state and wraps action dispatching in named functions for better developer experience. The provider makes both the current state and action functions available throughout the component tree, eliminating the need for prop drilling [12] while maintaining a single source of truth for application state.

3.3 Code Generation

Code generation is a key feature of SnapPy, translating the visual block structure into executable Python code in real-time.

3.3.1 Code Generation Architecture

The code generation system uses a visitor pattern to traverse the block structure:

```
export function generateCode(blocks: Block[]): string {
  const ctx: Context = { indent: 0, output: [] };

  for (const block of blocks) {
    visitBlock(ctx, block);
  }

  return ctx.output.join("\n");
}

function visitBlock(ctx: Context, block: Block) {
  switch (block.type) {
    case BlockType.Variable:
      return visitVariable(ctx, block);
    case BlockType.If:
      return visitIf(ctx, block);
    // Other block type handlers
  }
}
```

3.3.2 Indentation Management

A key aspect of Python code generation is proper indentation handling:

```
interface Context {
  indent: number;
  output: string[];
}

function getIndent(ctx: Context): string {
  return "    ".repeat(ctx.indent);
}

function addLine(ctx: Context, line: string): void {
  ctx.output.push(getIndent(ctx) + line);
}
```

This system uses Python's standard 4-space indentation, tracks the current indentation level as blocks are processed, and ensures nested blocks (like loops and conditionals) maintain proper indentation hierarchies.

3.3.3 Block-Specific Code Generation

Each block type has specialised code generation logic:

```
// Variable block code generation
function visitVariable(ctx: Context, block: VariableBlock) {
  let expressionCode = "";

  if (block.children.expression.length > 0) {
    expressionCode = visitExpression(ctx, block.children.expression);
  } else {
    expressionCode = "None";
  }

  addLine(ctx, `${block.selected} = ${expressionCode}`);
  return "";
}

// If block code generation
function visitIf(ctx: Context, block: IfBlock) {
  let conditionCode = "";
  if (block.children.condition.length > 0) {
    conditionCode = visitExpression(ctx, block.children.condition);
  } else {
    conditionCode = "True"; // Default condition if none provided
  }

  addLine(ctx, `if ${conditionCode}:`);
  ctx.indent++;

  // Process body blocks
  if (block.children.body.length === 0) {
    addLine(ctx, "pass");
  } else {
    for (const bodyBlock of sortBlocks(block.children.body)) {
      visitBlock(ctx, bodyBlock);
    }
  }

  ctx.indent--;
  return "";
}
```

3.3.4 Expression Handling

Expression blocks such as math operations are handled differently from statement blocks:

```
function visitMath(ctx: Context, block: MathBlock) {
  let leftCode = "";
  let rightCode = "";

  if (block.children.left.length > 0) {
    leftCode = visitExpression(ctx, block.children.left);
  }
  if (block.children.right.length > 0) {
    rightCode = visitExpression(ctx, block.children.right);
  }

  return `${leftCode} ${block.operator} ${rightCode}`;
}
```

Expression blocks return string values instead of adding lines to the output, can be nested to create complex expressions, and are composed together to build complete expressions that integrate with statement blocks.

3.4 Python Interpretation

SnapPy integrates a Python interpreter directly in the browser for executing code.

3.4.1 Pyodide Integration

Python execution is implemented using Pyodide[13], a WebAssembly-based[14] Python interpreter:

```
// Pyodide setup in a custom hook
useEffect(() => {
  async function loadPyodide() {
    try {
      setIsPyodideLoading(true);
      const { loadPyodide } = await import("pyodide");

      const pyodideInstance = await loadPyodide({
        indexURL: PYODIDE_CND,
        stdout: (line: string) => {
          setOutput((prev) => ({
            ...prev,
            error: false,
            message: [...prev.message, line],
          }));
        },
        stderr: (line: string) => {
          setOutput((prev) => ({
            ...prev,
            error: true,
            message: [...prev.message, line],
          }));
        },
      });

      setPyodide(pyodideInstance);
      setIsPyodideLoading(false);
    } catch (err) {
      console.error("Error loading Pyodide:", err);
      setIsPyodideLoading(false);
    }
  }

  loadPyodide();
}, []);
```

This integration enables client-side Python execution without server roundtrips, delivers immediate feedback on code execution and captures both standard output and error streams for display to the user.

3.4.2 Custom Hook for Code Execution

The Python execution is encapsulated in a custom React Hook[15]:

```
export default function useCodeEditor(
  canvas: Block[],
  entrypointBlockId: string | null,
) {
  const initialMessage = "# Snap a block to the start block to generate
      code!";

  const [pyodide, setPyodide] = useState<PyodideInterface>();
  const [isPyodideLoading, setIsPyodideLoading] = useState(true);
  const [code, setCode] = useState<string>(initialMessage);
  const [output, setOutput] = useState<Output>({ error: false, message:
      [] });
  const [isRunning, setIsRunning] = useState(false);

  // Effect for handling code generation when blocks change
  useEffect(() => {
    if (entrypointBlockId) {
      const startBlock = findBlockById(entrypointBlockId, canvas);
      if (!startBlock) {
        console.error(
          `Error in useCodeEditor: startBlock with id: ${
              entrypointBlockId} not found`,
        );
        return;
      }

      setCode(generateCode(getBlocksSequence(startBlock, canvas)));
    } else {
      setCode(initialMessage);
    }
  }, [entrypointBlockId, canvas]);

  // Load pyodide on first render
  useEffect(() => {
    async function loadPyodide() {
      // Pyodide loading logic
      // ...
    }
    loadPyodide();
  }, []);

  const handleCodeChange = (newCode: string): void => {
    setCode(newCode);
  };

  const runPython = async () => {
    // Code execution logic
    // ...
  };

  return {
    code,
    handleCodeChange,
    pyodide,
    isPyodideLoading,
    runPython,
    isRunning,
    output,
  };
}
```

This hook manages code generation from block structures, provides code editing capabilities, handles Python execution with output capture, and maintains loading states while properly handling errors.

3.4.3 Code Execution Process

The Python code execution process:

```
const runPython = async () => {
  if (!pyodide) {
    alert("Pyodide is not loaded yet");
    return;
  }

  setIsRunning(true);

  try {
    // Clear previous outputs
    setOutput({ error: false, message: [] });

    // Reset globals and run code
    const dict = pyodide.globals.get("dict");
    const globals = dict();
    const result = await pyodide.runPythonAsync(code, { globals:
        globals });
    globals.destroy();
    dict.destroy();

    // Add output to output message
    if (result && String(result)) {
      setOutput((prev) => ({ ...prev, message: [...prev.message, result
          ] }));
    }

    return result;
  } catch (err) {
    setOutput({ error: true, message: formatError(String(err)) });
    return;
  } finally {
    setIsRunning(false);
  }
};
```

This execution process creates an isolated Python environment for each run, executes the code, captures both standard output and return values, and presents errors in a user-friendly format while managing execution state for UI feedback.

3.4.4 Error Handling

Error handling is an important part of the Python execution system:

```
/**
 * Formats pyodide's error messages to make them more user-friendly.
 */
function formatError(error: string) {
  const lines = error.split("\n");
  let startLineIndex = 0;
  // Find the first line of the error message that's useful for the
     user
  for (let i = 0; i < lines.length; i++) {
    // The first line of the error message that matters starts with
       File "<exec>"
    if (lines[i].trim().startsWith(`File "<exec>"`)) {
      startLineIndex = i;
      break;
    }
  }

  const formattedError = lines.slice(startLineIndex);
  formattedError[0] = formattedError[0].split(",")[1].trim(); // Remove
      the initial part of the message
  return formattedError;
}
```

This error formatting extracts the most relevant parts of Python tracebacks, removes technical details that might confuse beginners, focuses on line numbers and clear error descriptions, and presents errors in a readable format that helps users identify and fix issues in their code.

3.5 Backend Implementation

3.5.1 Database Model

The PostgreSQL database schema includes:

```
model User {
  id        Int       @id @default(autoincrement())
  name      String
  email     String    @unique
  createdAt DateTime  @default(now()) @map("created_at")
  projects  Project[]
}

model Project {
  id        String @id @default(uuid())
  name      String
  User      User   @relation(fields: [userId], references: [id])
  userId    Int
  canvas    Json
  variables Json
}
```

3.5.2 Entity Relationship Diagram

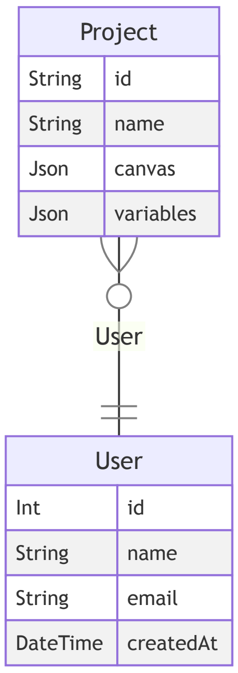


Figure 5: Entity Relationship Diagram

The schema employs a hybrid approach to data storage. The design maintains proper relational connections between users and projects through foreign key constraints, ensuring data consistency and enabling efficient queries across these entities. Meanwhile, it leverages PostgreSQL's native

JSON storage capability for the complex, hierarchical data structures that represent the canvas blocks and variables. This decision avoids the overhead of creating numerous relational tables to model deeply nested structures, which would introduce complexity from multiple joins. The JSON storage provides flexibility for evolving block structures while maintaining query performance, particularly for operations that need to retrieve or update entire project data at once.

3.5.3 API Design

The backend API provides a collection of database utility functions for managing users and their SnapPy projects:

```
// Fetch projects by user
export const fetchProjectsByUser = async (
  userId: number,
): Promise<Project[]> => {
  const projects = await prisma.project.findMany({
    where: { userId: userId },
  });
  return projects;
};

// Create a new project with initial blocks
export async function createProject(name: string): Promise<Project> {
  const user = await getUserSession();
  const project = await prisma.project.create({
    data: {
      name: name,
      userId: user.id,
      canvas: [
        {
          id: "start",
          type: BlockType.ProgramStart,
          shape: BlockShape.Square,
          coords: { x: 400, y: 200 },
          isWorkbenchBlock: false,
          state: BlockState.Idle,
          parentId: null,
          prevId: null,
          nextId: null,
          children: null,
        },
      ],
      variables: ["x"],
    },
  });
  return project;
}

```

```
// Handle JSON serialization for project updates
export async function updateProject(
  projectId: string,
  canvas: Block[],
  variables: string[],
): Promise<Project> {
  const canvasJson: Prisma.InputJsonValue = JSON.parse(JSON.stringify(
     canvas));
  const variablesJson: Prisma.InputJsonValue = JSON.parse(
    JSON.stringify(variables),
  );
  const project = await prisma.project.update({
    where: { id: projectId },
    data: { canvas: canvasJson, variables: variablesJson },
  });
  return project;
}
```

These functions handle the core operations of fetching, creating, and updating projects with appropriate JSON serialization for complex data structures. The API integrates with Prisma ORM for type-safe database operations while maintaining proper user authentication through session management.

4 Problems Solved

4.1 Block Rendering Approach

Problem: Choosing an appropriate rendering method for blocks that would support nesting and interactive elements.
Solution: After evaluating multiple approaches, we decided on HTML/CSS-based blocks rather than SVG-based blocks.

HTML/CSS approach advantages

- The browser handles the complexities of positioning and nesting
- Native support for interactive elements like input fields and select boxes
- Seamless integration with React component model

Trade-offs:

- Visual simplicity compared to custom SVG designs
- Less precise control over appearance
- Reliance on browser rendering behavior

We chose to prioritise functional reliability and implementation speed over visual complexity. This allowed us to focus resources on the core user experience and block interaction logic.

The resulting implementation provides a flexible foundation that can be visually enhanced in future iterations while maintaining the robust interaction model necessary for a visual programming environment.

4.2 Canvas Data Structure Evolution

Problem: Designing a data structure that could represent the complex relationships between blocks on the canvas.
Solution: The canvas data structure evolved through several iterations:

1. **Initial Implementation:** Simple array of blocks
2. **Adding Hierarchy:** Array of tree structures to represent nested blocks
3. **Final Approach:** Hybrid model with both sequential and hierarchical relationships
    - Blocks track `prevId` and `nextId` for sequential connections
    - Blocks track `parentId` and `children` for hierarchical relationships
    - This dual-relationship system enables representing complex code structures visually

This approach allows SnapPy to model the full range of Python programming constructs, from simple sequences to deeply nested control structures.

4.3 Drag and Drop Implementation

Problem: Implementing a drag and drop system that supports both block movement and hierarchical nesting while maintaining proper constraints.
Solution:

- Created a custom implementation using the dnd-kit library
- Developed specialised event handlers for different drag scenarios
- Implemented drag monitors to manage active dragging states
- Built custom pointer sensors to handle special cases

4.4 Minimising Runtime Errors

Problem: Building a complex application with numerous component interactions and state transformations increases the risk of runtime errors.
Solution: We adopted TypeScript for its robust type system, which proved invaluable throughout development.

- Created a comprehensive set of interfaces and type definitions for blocks
- Used union types to handle the various block shapes and behaviors
- Leveraged TypeScript's type checking to catch potential errors at compile time
- Enabled better IDE support for code navigation and refactoring
- Facilitated self-documenting code through explicit type declarations

TypeScript's static typing dramatically reduced the incidence of runtime errors, especially when manipulating complex block structures. The additional development time spent on type definitions was quickly recovered through faster debugging and fewer unexpected behaviors.

4.5 Unfamiliar Technical Territory

Problem: Many aspects of this project were new to us. We had never built a UI with such complex interactions, never implemented drag and drop functionality, and never integrated so many different components into a single interactive canvas.
Solution:

- Conducted extensive prototyping to de-risk technical challenges before full implementation
- Built isolated proof-of-concept implementations for critical features
- Developed an MVP with all core functions before expanding features
- Used iterative development with frequent testing to validate each component
- Maintained clear separation of concerns to manage complexity by continuously refactoring both components and architecture

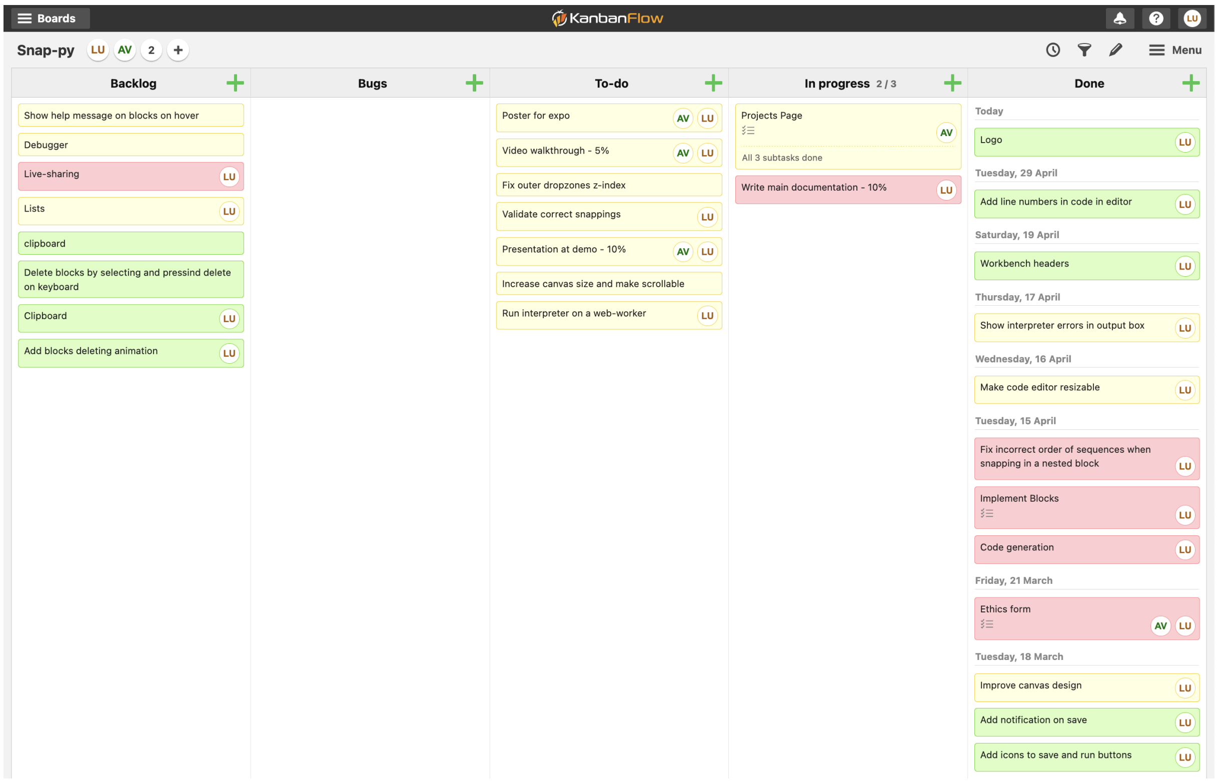
4.6 Finding an Appropriate Workflow

Problem: Developing a novel application with complex interactions required extensive prototyping while maintaining an acceptable coding quality.
Solution:

- Implemented a comprehensive CI/CD pipeline with static analysis tools, ESLint for code style enforcement, and Jest for automated testing
- Established branch protection on main to prevent direct pushes and require peer reviews
- Created a merge policy requiring all pipeline checks to pass before pull requests could be merged
- Adopted trunk-based development with frequent small commits to minimize integration conflicts
- Embraced Extreme Programming's philosophy, prioritizing working software and iterative refinement

- Utilized a structured Kanban board with dedicated columns for Backlog, Bugs, To Do, In Progress and Done to visualize workflow to manage priorities

```yaml
# Our GitLab pipeline

image: node:23-alpine

stages:
  - lint
  - test

lint:
  stage: lint
  script:
    - cd src
    - npm install -g bun@1.1.39
    - bun install
    - bun run lint
    - bun run format:check

test:
  stage: test
  script:
    - cd src
    - npm install -g bun@1.1.39
    - bun install
    - bun run test
```



Snap-py LU AV 2

Backlog
- Show help message on blocks on hover
- Debugger
- Live-sharing LU
- Lists LU
- clipboard
- Delete blocks by selecting and pressind delete on keyboard
- Clipboard LU
- Add blocks deleting animation LU

Bugs

To-do
- Poster for expo AV LU
- Video walkthrough - 5% AV LU
- Fix outer dropzones z-index
- Validate correct snappings LU
- Presentation at demo - 10% AV LU
- Increase canvas size and make scrollable
- Run interpreter on a web-worker LU

In progress 2 / 3
- Projects Page AV — All 3 subtasks done
- Write main documentation - 10% LU

Done

Today
- Logo LU

Tuesday, 29 April
- Add line numbers in code in editor LU

Saturday, 19 April
- Workbench headers LU

Thursday, 17 April
- Show interpreter errors in output box LU

Wednesday, 16 April
- Make code editor resizable LU

Tuesday, 15 April
- Fix incorrect order of sequences when snapping in a nested block LU
- Implement Blocks LU
- Code generation LU

Friday, 21 March
- Ethics form AV LU

Tuesday, 18 March
- Improve canvas design LU
- Add notification on save LU
- Add icons to save and run buttons LU

Figure 6: Kanban board

5 Results

5.1 Testing Strategy

SnapPy was tested using a comprehensive strategy that follows industry best practices [16]:

- Unit tests for individual components and functions
- Integration tests for system interactions
- Component tests for the UI components

5.2 Testing Tools

Testing was implemented using:

- Jest [17] for JavaScript/TypeScript unit and integration testing
- React Testing Library [18] for component testing

Jest provided the foundation for test execution and assertions throughout the codebase. React Testing Library supplemented this with component-specific testing patterns that focus on user interactions rather than implementation details.

5.3 Block System Tests

Extensive testing was performed on the block system to ensure reliability:

5.3.1 Block Manipulation Tests

Tests were created to verify the most important block operations:

```
test("SELECT_BLOCK should mark a block as selected and update
    selectedBlockId", () => {
  const action: CanvasAction = {
    type: CanvasEvent.SELECT_BLOCK,
    payload: { id: "block1" },
  };

  const newCanvas = BlocksReducer(initialCanvas, action);

  expect(newCanvas.selectedBlockId).toBe("block1");
  expect(newCanvas.canvas.find((b) => b.id === "block1")!.state).toBe(
    BlockState.Selected,
  );
});
```

This test verifies the state transitions when selecting a block, ensuring both the global selection state and the block’s internal state are updated correctly.

5.3.2 Block Connection Tests

Comprehensive tests ensure blocks connect correctly in all scenarios:

```
test("Snaps single block above single block correctly", () => {
  const action: CanvasAction = {
    type: CanvasEvent.SNAP_BLOCK,
    payload: {
      id: "block1",
      targetId: "block2",
      position: OuterDropzonePosition.Top,
    },
  };

  const testCanvas = { ...initialCanvas };
  const newCanvas = BlocksReducer(testCanvas, action);
  const newBlock1 = findBlockById("block1", newCanvas.canvas)!;
  const newBlock2 = findBlockById("block2", newCanvas.canvas)!;

  const expectedBlock1 = {
    ...block1,
    nextId: "block2",
    coords: { x: 200, y: 200 },
  };
  const expectedBlock2 = {
    ...block2,
    prevId: "block1",
  };

  expect(newBlock1).toEqual(expectedBlock1);
  expect(newBlock2).toEqual(expectedBlock2);
});
```

This test validates that blocks correctly establish sequential relationships when connected, with correct updates to position coordinates and bidirectional linkage between blocks.
In SnapPy there are 10 different ways of snapping blocks together, all updating state differently. This part of the codebase is particularly robust, as it was developed by using test-driven development.

5.3.3 Nested Block Tests

Tests ensure that complex nested structures are handled correctly:

```
test("Nests sequence of blocks with nested blocks correctly", () => {
  const sequenceBlock1: EmptyBlock = { ...block1, nextId: "block2" };
  const sequenceBlock2: EmptyBlock = {
    ...block1,
    id: "block2",
    prevId: "block1",
    nextId: "block3",
  };
  const sequenceBlock3: VariableBlock = {
    ...block3,
    id: "block3",
    prevId: "block2",
    children: {
      expression: [
        {
          ...block1,
          id: "block5",
          state: BlockState.Nested,
          parentId: "block3",
        },
      ],
    },
  };
  const parent = { ...block4 };

  const testCanvas: CanvasState = {
    ...initialCanvas,
    canvas: [sequenceBlock1, sequenceBlock2, sequenceBlock3, parent],
  };

  let newCanvas = BlocksReducer(testCanvas, {
    type: CanvasEvent.ADD_CHILD_BLOCK,
    payload: {
      id: "block1",
      targetId: "block4",
      prefix: "body",
    },
  });

  // Assertions to verify correct nesting behavior
  // ...
});
```

This test addresses the crucial handling of hierarchical block relationships, where blocks can contain other blocks or entire sequences. The test sets up a complex initial state with a linked sequence of blocks (some of which already contain nested elements) and then tests the operation of nesting this entire sequence into another parent block. This validates that the system correctly preserves both the sequential relationships within the nested group and establishes the proper parent-child relationship with the container block.

5.4 Utility Function Tests

Utility functions are crucial for block management and were extensively unit tested:

```
describe("findBlockById", () => {
  test("Returns null for empty canvas", () => {
    const emptyCanvas: Block[] = [];
    expect(findBlockById("id", emptyCanvas)).toBeNull();
  });

  test("Finds non-nested block", () => {
    const foundBlock = findBlockById("block2", flatCanvas);
    expect(foundBlock).not.toBeNull();
    expect(foundBlock).toMatchObject({
      id: "block2",
    });
  });

  test("Finds nested blocks", () => {
    const foundDeepBlock = findBlockById("target-block-deep",
        nestedCanvas);
    expect(foundDeepBlock).not.toBeNull();
    expect(foundDeepBlock).toMatchObject({
      id: "target-block-deep",
    });
  });
});
```

These tests verify that utility functions like `findBlockById` correctly handle various edge cases including empty collections, flat block structures, and deeply nested hierarchies. This testing approach ensures that the core functions that traverse and manipulate the block structure remain reliable across all usage scenarios, preventing subtle bugs that might otherwise arise from edge cases in the block traversal logic.

5.5 Code Generation Tests

Tests ensure that Python code is correctly generated from block structures:

```
describe("Code generation", () => {
  test("Generates variable with no expression correctly", () => {
    // expected: x = None
    const blocks: Block[] = [{ ...varBlock }];
    expect(generateCode(blocks)).toBe("x = None");
  });

  test("Generates variable with simplest numeric expression correctly",
      () => {
    // expected: x = 1
    const blocks: Block[] = [
      {
        ...varBlock,
        children: { expression: [{ ...numBlock }] },
      },
    ];
    expect(generateCode(blocks)).toBe("x = 1");
  });

  test("Generates variable with nested complex numeric expression
      correctly", () => {
    // expected: x = 1 + 2 + 3 + 4
    const leftAddition: MathBlock = {
      ...additionBlock,
      children: {
        left: [{ ...numBlock }],
        right: [{ ...numBlock, value: "2" }],
      },
    };
    const rightAddition: MathBlock = {
      ...additionBlock,
      children: {
        left: [{ ...numBlock, value: "3" }],
        right: [{ ...numBlock, value: "4" }],
      },
    };
    const blocks: Block[] = [
      {
        ...varBlock,
        children: {
          expression: [
            {
              ...additionBlock,
              children: {
                left: [leftAddition],
                right: [rightAddition],
              },
            },
          ],
        },
      },
    ];
    expect(generateCode(blocks)).toBe("x = 1 + 2 + 3 + 4");
  });
});
```

These tests verify the code generation system's ability to translate blocks into syntactically correct Python code. Starting with simple variable assignments, the tests progress to increasingly complex expressions with nested operations. The tests ensure correct handling of default values when expressions are missing, proper generation of simple expressions, and accurate composition of complex nested expressions with appropriate operator precedence.

5.6 UI Component Tests

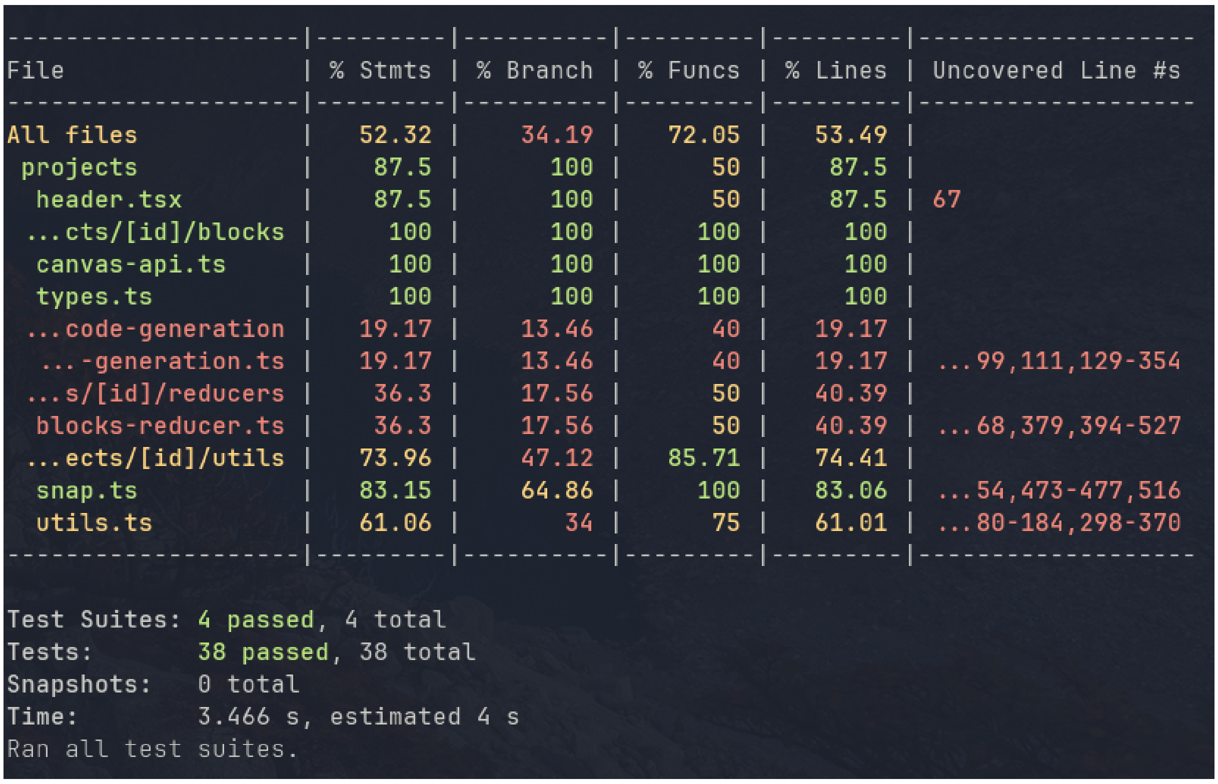
UI components were tested for correct rendering and user interaction:

```

describe('Header', () => {
  const session: Session = {
    name: 'John Snow',
    email: 'john@snow.ie',
    image: 'someurl',
    id: 1,
  };

```

```
describe('Header', () => {
  it('displays correct email when menu is opened', async () => {
    render(<Header session={session} />);
    const user = userEvent.setup();
    await user.click(screen.getByRole('button'));
    // Wait for menu to appear
    await waitFor(() => {
      const email = screen.getByText(/john@snow\.ie/i);
      expect(email).toBeInTheDocument();
    });
  });

  it('displays avatar placeholder if user image fails to load', async
     () => {
    render(<Header session={{ ...session, image: undefined }} />);
    const user = userEvent.setup();
    await user.click(screen.getByRole('button'));
    // Wait for menu to appear
    await waitFor(() => {
      const avatarPlaceholder = document.querySelector(
        '.mantine-Avatar-placeholder'
      );
      expect(avatarPlaceholder).toBeInTheDocument();
    });
  });
});
```

We render component trees in a simplified test environment provided by React Testing Library. This approach allows us to validate component behavior without coupling tests to implementation details. The tests focus on user interactions and verify that components respond correctly to those interactions. We use accessibility-focused selectors like getByRole and text content patterns, we ensure tests remain robust against cosmetic changes while confirming that components meet accessibility requirements. The asynchronous testing pattern with waitFor properly handles component updates that occur after user actions.

5.7 Test Coverage



```
--------------------|---------|----------|---------|---------|-------------------
File                | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
--------------------|---------|----------|---------|---------|-------------------
All files           |   52.32 |    34.19 |   72.05 |   53.49 |
 projects           |    87.5 |      100 |      50 |    87.5 |
  header.tsx        |    87.5 |      100 |      50 |    87.5 | 67
 ...cts/[id]/blocks |     100 |      100 |     100 |     100 |
  canvas-api.ts     |     100 |      100 |     100 |     100 |
  types.ts          |     100 |      100 |     100 |     100 |
 ...code-generation |   19.17 |    13.46 |      40 |   19.17 |
  ...-generation.ts |   19.17 |    13.46 |      40 |   19.17 | ...99,111,129-354
 ...s/[id]/reducers |    36.3 |    17.56 |      50 |   40.39 |
  blocks-reducer.ts |    36.3 |    17.56 |      50 |   40.39 | ...68,379,394-527
 ...ects/[id]/utils |   73.96 |    47.12 |   85.71 |   74.41 |
  snap.ts           |   83.15 |    64.86 |     100 |   83.06 | ...54,473-477,516
  utils.ts          |   61.06 |       34 |      75 |   61.01 | ...80-184,298-370
--------------------|---------|----------|---------|---------|-------------------

Test Suites: 4 passed, 4 total
Tests:       38 passed, 38 total
Snapshots:   0 total
Time:        3.466 s, estimated 4 s
Ran all test suites.
```

Figure 7: Jest's test coverage output

The block system received comprehensive test coverage due to its complexity and central importance. We developed extensive tests for block manipulation, connections, and nested structures to ensure reliability across all possible user interactions. The utility functions that traverse and manipulate block structures were also rigorously tested to prevent subtle bugs in these critical operations. We chose to sacrifice coverage on certain aspects of the application, such as the UI components and code generation, as these are simpler parts and bugs are easier to spot simply while using the application. However, we aim at improving test coverage in these areas by the time of the demo to ensure the application meets quality standards across all its functionality.

6 Future Work

Based on user feedback, technical evaluation and research in programming education, we have identified several key areas for the future development of SnapPy:

6.1 Enhanced Block Types

Adding support for functions and data structures would significantly expand SnapPy's educational scope, allowing students to learn more advanced programming concepts while maintaining the visual scaffolding.

Implementation challenges:

- Designing intuitive visual representations for function parameters and return values
- Creating a balance between simplicity and functionality for data structure manipulation
- Maintaining visual clarity when representing abstract concepts
- Implementing appropriate code generation for these more complex structures

6.2 Advanced Debugging Tools

A visual debugger would transform SnapPy from a coding environment into a powerful learning tool by making program execution transparent and concrete for novice programmers.

Implementation challenges:

- Instrumenting Python code for step-by-step execution in Pyodide
- Synchronizing the visual canvas with execution state
- Capturing and displaying variable values at each step
- Creating an intuitive debugging interface that doesn't overwhelm beginners
- Visualizing complex program state in a comprehensible manner

6.3 Real-Time Collaboration

Real-time editing would enable pair programming and classroom activities.

Implementation challenges:

- Conflict resolution for simultaneous changes
- Maintaining consistent state across clients
- Handling network delays and disconnections

6.4 Mobile Support

Extending SnapPy to mobile devices would significantly increase accessibility, allowing programming practice on tablets in classrooms or on-the-go learning.

Implementation challenges:

- Redesigning the drag-and-drop interface for touch interaction
- Accommodating varying screen sizes while maintaining usability
- Maintaining feature parity between desktop and mobile versions

References

[1] "SPA (Single-page application) - MDN Web Docs Glossary: Definitions of Web-related terms | MDN." https://developer.mozilla.org/en-US/docs/Glossary/SPA, Dec. 2024.

[2] "Next.js by Vercel - The React Framework." https://nextjs.org/.

[3] R. Weeda, Smetsers, and E. and Barendsen, "Unraveling novices' code composition difficulties," *Computer Science Education*, vol. 34, no. 3, pp. 414–441, Jul. 2024, doi: 10.1080/08993408.2023.2169067.

[4] "To block or not to block, that is the question | Proceedings of the 14th International Conference on Interaction Design and Children." https://dl.acm.org/doi/10.1145/2771839.2771860.

[5] V. Cutting and N. Stephen, "A Review on using Python as a Preferred Programming Language for Beginners," vol. 8, pp. 4258–4263, Aug. 2021.

[6] "Scratch - Imagine, Program, Share." https://scratch.mit.edu/.

[7] "Microsoft MakeCode for micro:bit," *Microsoft MakeCode for micro:bit*. https://makecode.microbit.org/.

[8] "Dnd kit – a modern drag and drop toolkit for React." https://dndkit.com/.

[9] "Prisma | Simplify working and interacting with databases," *Prisma*. https://www.prisma.io.

[10] "Extracting State Logic into a Reducer – React." https://react.dev/learn/extracting-state-logic-into-a-reducer.

[11] "Reading 9: Mutability & Immutability." https://web.mit.edu/6.005/www/fa15/classes/09-immutability/.

[12] "Passing Data Deeply with Context – React." https://react.dev/learn/passing-data-deeply-with-context.

[13] "Pyodide — Version 0.27.5." https://pyodide.org/en/stable/.

[14] "WebAssembly." https://webassembly.org/.

[15] "Reusing Logic with Custom Hooks – React." https://react.dev/learn/reusing-logic-with-custom-hooks.

[16] "Testing Overview – React." https://legacy.reactjs.org/docs/testing.html.

[17] "Jest." https://jestjs.io/.

[18] "React Testing Library | Testing Library." https://testing-library.com/docs/react-testing-library/intro/, Jun. 2024.