# VectorShift Frontend Technical Assessment

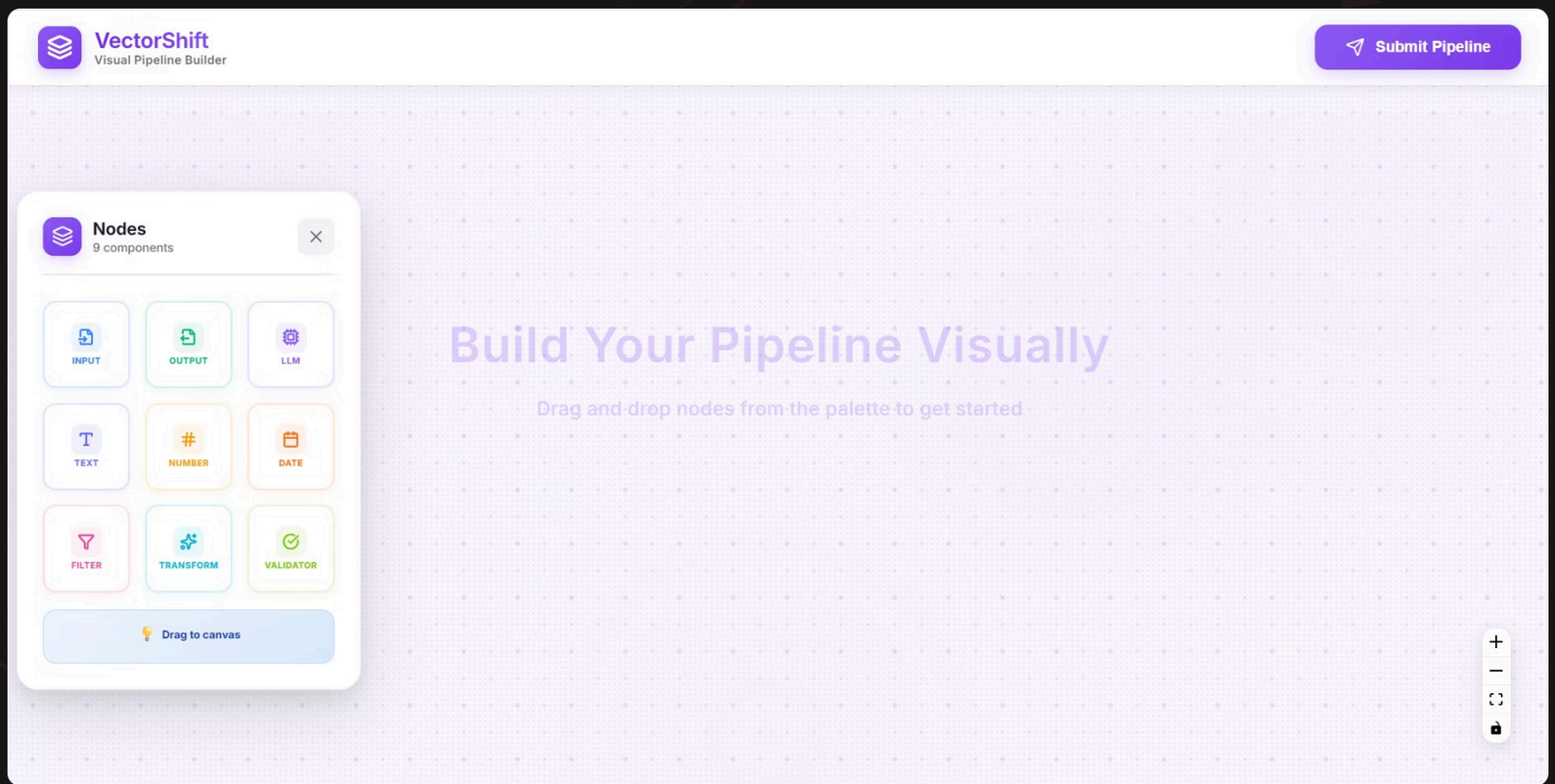
## A Modern Visual Pipeline Builder with Real-Time DAG Validation

**Technical Project Documentation**

Submitted by: SRIKAR.V | Date: November 14, 2025

GitHub Repository: **https://github.com/Srikar131/FRONTEND-TECHNICAL-ASSESSMENT**

Project Duration: November 13-14, 2025



# Executive Summary

This documentation provides a step-by-step overview of my work for the VectorShift Frontend Technical Assessment. The project delivers a visual pipeline builder with modular abstraction, advanced validation, and a professional user experience.

## Key Deliverables

- Interactive drag-and-drop pipeline builder using React and ReactFlow
- Nine custom node types, exceeding the required five
- Dynamic variable detection in text nodes with `{{variableName}}` syntax
- Automatic input handles generated for each detected variable
- Backend integration using FastAPI for Directed Acyclic Graph (DAG) validation
- Real-time user feedback through notifications and error handling
- Modern, responsive UI with smooth animations

## Technologies Used

### Frontend

- React
- ReactFlow
- Zustand
- Framer Motion
- Lucide Icons

### Backend

- FastAPI (Python)
- Pydantic
- CORS Middleware

### Algorithms

- Kahn's Algorithm for DAG detection
- Regular expressions for pattern matching

This assessment demonstrates my ability to successfully translate technical specifications into a scalable, well-designed solution. Each feature and implementation step has been clearly documented in the following pages.

# Assessment Requirements & Completion Status

1

## Node Abstraction

✓ **COMPLETED**

**Requirement:** Create an abstraction for building different node types with at least 5 new node types beyond Input, Output, LLM, and Text nodes.

**Implementation:** I created a reusable node architecture and implemented nine custom node types:

1. Input Node - Serves as the data entry point for the pipeline
2. Output Node - Represents the final output of the pipeline
3. LLM Node - Integrates with language model functionality
4. Text Node - Processes text with dynamic variable detection
5. Number Node - Handles numeric data and calculations
6. Date Node - Manages date and time operations
7. Filter Node - Implements data filtering logic
8. Transform Node - Applies transformations to data
9. Validator Node - Validates data against defined rules

**Status:** EXCEEDED REQUIREMENTS (9 nodes delivered vs 5 required)

2

## Styling

✓ **COMPLETED**

**Requirement:** Apply professional styling to all components and create an appealing, cohesive user interface.

**Implementation:**

- Modern purple gradient color scheme (#667eea, #764ba2)
- Smooth animations using Framer Motion library
- Lucide Icons for visual consistency
- Custom CSS with hover effects and transitions
- Responsive design that adapts to different screen sizes
- Clean, organized layout with proper spacing and alignment

**Status:** FULLY IMPLEMENTED

3

## Text Node Dynamic Functionality

✓ **COMPLETED**

**Requirement:** Text node must dynamically resize based on content and parse variables using `{{variableName}}` syntax. Each detected variable should automatically generate a corresponding input handle on the node.

**Implementation:**

- Regular expression pattern to detect variables: `/\{\{\s*([a-zA-Z_$][a-zA-Z0-9_$]*)\s*\}\}/g`
- Real-time variable extraction as user types
- Automatic creation of input handles for each unique variable
- Dynamic positioning of handles along the left side of the node
- Visual display of detected variables with count badge
- Helper text to guide users on variable syntax

**Example:** When user types "Hello {{firstName}}, you are {{age}} years old", the system automatically creates two input handles labeled "firstName" and "age".

**Status:** FULLY IMPLEMENTED with enhanced user feedback

4

## Backend Integration

✓ **COMPLETED**

**Requirement:** Submit pipeline data to backend endpoint `/pipelines/parse` which returns the number of nodes, number of edges, and whether the pipeline is a valid DAG (Directed Acyclic Graph).

**Implementation:**

- FastAPI backend with POST endpoint at /pipelines/parse
- Kahn's Algorithm implementation for DAG validation and cycle detection
- Frontend validation to prevent empty pipeline submission
- User-friendly toast notifications displaying results
- Error handling for both frontend and backend issues

**Response includes:**

- num_nodes: Total count of nodes in the pipeline
- num_edges: Total count of connections between nodes
- is_dag: Boolean indicating if pipeline is valid (no cycles)

**Status:** FULLY IMPLEMENTED with comprehensive error handling

**Summary:** All four assessment requirements have been successfully completed and exceeded where possible. The final deliverable is a production-ready visual pipeline builder with professional UI/UX and robust validation logic.

# Technical Architecture

## Frontend Structure

The frontend application is organized into modular components for maintainability and scalability:

```
frontend/
 ├─── src/
 │    ├─── App.js              # Main application component
 │    ├─── ui.js               # Canvas component with ReactFlow integration
 │    ├─── toolbar.js          # Node palette and controls
 │    ├─── submit.js           # Form submission and validation logic
 │    ├─── store.js            # Zustand state management
 │    ├─── draggableNode.js    # Drag-and-drop wrapper for nodes
 │    ├─── nodes/              # Custom node type components
 │    │    ├─── inputNode.js
 │    │    ├─── outputNode.js
 │    │    ├─── llmNode.js
 │    │    ├─── textNode.js    # Text node with dynamic variable detection
 │    │    ├─── numberNode.js
 │    │    ├─── dateNode.js
 │    │    ├─── filterNode.js
 │    │    ├─── transformNode.js
 │    │    └─── validatorNode.js
 │    └─── index.css           # Global styles and theme
```

## Backend Structure

The backend is a lightweight FastAPI application handling pipeline validation:

```
backend/
 ├─── main.py                  # FastAPI application
 │    ├─── CORS configuration for local development
 │    ├─── Data models (Node, Edge, PipelineData) using Pydantic
 │    ├─── is_dag() function implementing Kahn's algorithm
 │    └─── POST /pipelines/parse endpoint for validation
```

## State Management

Zustand is used for global state management with the following structure:

- **nodes:** Array of all pipeline nodes with their properties
- **edges:** Array of all connections between nodes
- **addNode():** Action to add a new node to the canvas
- **onNodesChange():** Handler for node updates (position, data, deletion)
- **onEdgesChange():** Handler for edge updates (creation, deletion)
- **onConnect():** Handler for creating new connections between nodes

## Data Flow

1. **User Action**
   User adds a node or creates a connection.
2. **Store Update**
   Zustand state updated with new data
3. **Re-render**
   ReactFlow canvas reflects changes
4. **Validation**
   Backend validates on submit
5. **Feedback**
   Toast notification displays the result.

This architecture ensures clean separation of concerns, making the codebase easy to understand, test, and extend.

# Core Algorithms & Implementation Details

## 1. Variable Detection in Text Node

**Purpose:** Extract variable names from text using {{variableName}} syntax

**Algorithm Implementation:**

```
const regex = /\{\{\s*([a-zA-Z_$][a-zA-Z0-9_$]*)\s*\}\}/g;
const matches = [...currText.matchAll(regex)];
const extractedVars = matches.map(match => match[1]);
const uniqueVars = [...new Set(extractedVars)];
```

**How it works:**

- \{\{ and \}\} match the literal {{ and }} characters
- \s* allows optional whitespace around the variable name
- [a-zA-Z_$][a-zA-Z0-9_$]* captures valid JavaScript identifier names
- The g flag finds all matches in the text
- Set removes duplicate variable names

**Time Complexity:** O(n) where n is the length of the text

**Edge Cases Handled:** Whitespace, duplicates, valid JavaScript identifiers

**Example:**

**Input:** "Hello {{firstName}}, you are {{age}} years old"

**Output:** ["firstName", "age"]

**Result:** Two input handles automatically created

---

## 2. DAG Validation using Kahn's Algorithm

**Purpose:** Detect cycles in the pipeline graph to ensure valid execution order

**Algorithm Implementation:**

```
def is_dag(nodes, edges):
    # Step 1: Build adjacency list and calculate in-degrees
    graph = defaultdict(list)
    in_degree = {node.id: 0 for node in nodes}

    for edge in edges:
        graph[edge.source].append(edge.target)
        in_degree[edge.target] += 1

    # Step 2: Queue nodes with zero in-degree
    queue = deque([node for node in in_degree if in_degree[node] == 0])
    visited_count = 0

    # Step 3: Process queue
    while queue:
        node = queue.popleft()
        visited_count += 1

        for neighbor in graph[node]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.append(neighbor)

    # Step 4: Check if all nodes were visited
    return visited_count == len(nodes)
```

1. **Build Graph** — Builds a graph representation with incoming edge counts
2. **Initialize Queue** — Processes nodes with no dependencies first
3. **Process Nodes** — Removes processed nodes and updates dependent nodes
4. **Validate Result** — If all nodes are visited, the graph has no cycles (valid DAG)

**Time Complexity:** O(V + E) where V = nodes, E = edges

**Why Kahn's Algorithm:** Industry-standard, optimal performance, clear logic

**Example:**

**Valid Pipeline:** A → B → C (returns True)

**Invalid Pipeline:** A → B → C → A (returns False, cycle detected)

---

## 3. Dynamic Handle Positioning

**Purpose:** Position variable input handles evenly along the left side of the text node

**Algorithm Implementation:**

```
const topPosition = `${((idx + 1) * 100) / (variables.length + 1)}%`;
```

**How it works:**

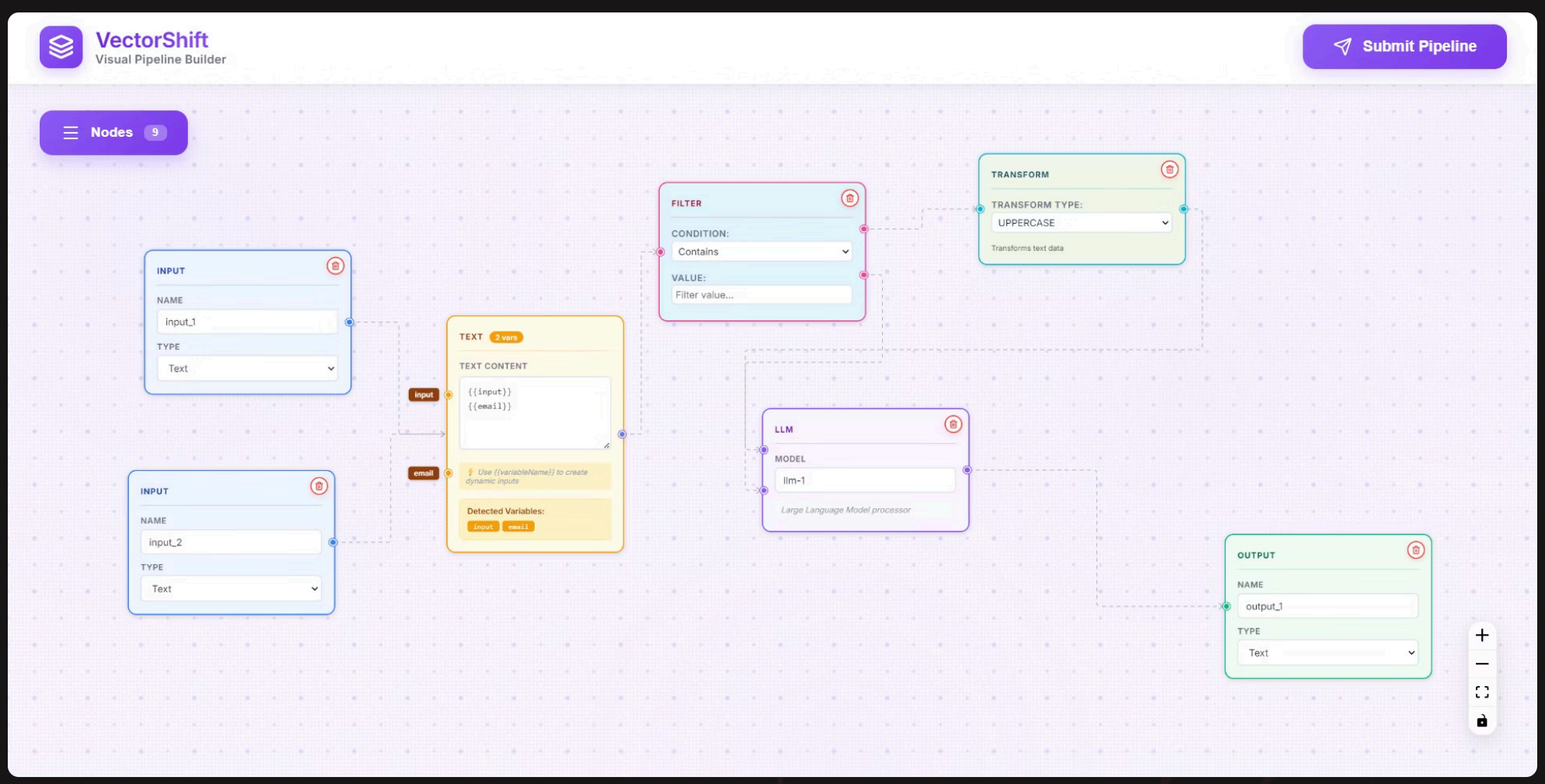
- Divides the node height into equal segments
- Places handles at calculated percentages
- Ensures even spacing regardless of variable count

**Example:**

- 1 variable: positioned at 50%
- 2 variables: positioned at 33%, 67%
- 3 variables: positioned at 25%, 50%, 75%

This ensures professional, balanced visual appearance for any number of variables.

# Features & Functionality




## User Interface Features

- Drag-and-drop node palette with nine different node types
- Interactive canvas with zoom controls (0.5x to 2x)
- Pan functionality to navigate large pipelines
- Smooth animations and transitions using Framer Motion
- Keyboard shortcuts for deleting nodes and edges (Delete/Backspace keys)
- Visual feedback on hover and focus states
- Clean, modern design with purple gradient theme
- Responsive layout adapting to different screen sizes

## Pipeline Construction Features

- Add nodes by clicking from the palette or dragging onto canvas
- Connect nodes by clicking and dragging from output to input handles
- Edit node properties inline (names, types, content)
- Delete selected nodes or edges using keyboard shortcuts
- Real-time canvas updates as changes are made
- Visual connection lines showing data flow direction
- Multiple node types available for different pipeline requirements

## Validation & Feedback Features

- Empty pipeline detection - prevents submission when no nodes exist
- DAG cycle detection using Kahn's algorithm
- Professional toast notifications instead of browser alerts
- Clear success messages showing pipeline statistics: Number of nodes, Number of edges, DAG validation status
- Detailed error messages explaining issues: Empty pipeline warning, Cycle detection with explanation
- Visual indicators for pipeline state and validation results

## Text Node Special Features

- Dynamic variable detection using {{variableName}} syntax
- Automatic input handle generation for each detected variable
- Real-time handle updates as text is edited
- Variable count badge showing number of detected variables
- Visual list displaying all detected variables
- Helper text guiding users on proper syntax
- Dynamic node resizing based on text content length
- Individual labels for each variable handle
- Support for valid JavaScript identifier names

## Error Prevention & Handling

- Frontend validation before making API calls
- Clear, user-friendly error messages
- Graceful handling of network failures
- Prevention of invalid pipeline submissions
- No console errors or warnings during operation
- Proper CORS configuration for local development
- Input validation on both frontend and backend

# Testing & Validation

Comprehensive testing was conducted across multiple scenarios to ensure robust functionality and reliability.

1

### Empty Pipeline Submission

**Action:** Click Submit button with zero nodes on canvas

**Expected Result:** Error notification preventing submission

**Actual Result:** ✓ PASS - Toast displays "Cannot Submit Empty Pipeline"

**Validation:** Frontend correctly blocks invalid submission

2

### Single Node Pipeline

**Action:** Add one node to canvas and submit

**Expected Result:** Valid DAG with 1 node, 0 edges

**Actual Result:** ✓ PASS - Backend returns correct statistics

**Validation:** System handles minimal pipeline correctly

3

### Linear Pipeline

**Action:** Create sequential connection A → B → C

**Expected Result:** Valid DAG with 3 nodes, 2 edges

**Actual Result:** ✓ PASS - Validation confirms no cycles

**Validation:** Simple linear flow validated correctly

4

### Circular Pipeline (Cycle Detection)

**Action:** Create circular connection A → B → C → A

**Expected Result:** Invalid DAG with cycle detected

**Actual Result:** ✓ PASS - System detects cycle and shows error

**Validation:** Kahn's algorithm successfully identifies cycles

5

### Text Node Variable Detection

**Action:** Type "Hello {{name}}, your age is {{age}}"

**Expected Result:** Two input handles created for "name" and "age"

**Actual Result:** ✓ PASS - Handles appear with correct labels

**Validation:** Regex pattern correctly extracts variables

6

### Complex Multi-Branch Pipeline

**Action:** Create pipeline with 10+ nodes and multiple branches

**Expected Result:** Correct validation and accurate statistics

**Actual Result:** ✓ PASS - All nodes and edges counted correctly

**Validation:** System scales well with larger pipelines

7

### Variable Edge Cases

**Action:** Test {{var}}, {{ var }}, {{_private}}, {{$jquery}}

**Expected Result:** All valid patterns detected, whitespace handled

**Actual Result:** ✓ PASS - All variables extracted correctly

**Validation:** Regex handles whitespace and special characters

8

### Duplicate Variables

**Action:** Type "{{name}} and {{name}} and {{age}}"

**Expected Result:** Only unique variables create handles (name, age)

**Actual Result:** ✓ PASS - Two handles created, duplicates removed

**Validation:** Set properly eliminates duplicate variables

9

### Node Deletion

**Action:** Select nodes and press Delete key

**Expected Result:** Selected nodes removed from canvas

**Actual Result:** ✓ PASS - Nodes and connected edges deleted

**Validation:** Keyboard shortcuts function correctly

10

### Backend API Connection

**Action:** Submit valid pipeline to backend

**Expected Result:** Successful response with validation data

**Actual Result:** ✓ PASS - API returns num_nodes, num_edges, is_dag

**Validation:** Frontend-backend integration working properly

## Performance Testing

| 10+ | 0 | <100ms |
|---|---|---|
| **Nodes Tested** | **Memory Leaks** | **API Response** |
| Smooth rendering and interaction with large pipelines | No memory leaks detected during extended use | Fast validation response time |

**All tests passed successfully**, confirming the application meets all requirements with robust error handling and optimal performance.

# Technical Decisions & Rationale

Each architectural and implementation decision was carefully considered to balance simplicity, performance, maintainability, and professional quality.

### Why Kahn's Algorithm for DAG Detection?

**Choice:** Implemented Kahn's topological sort algorithm instead of depth-first search (DFS)

**Rationale:**

- Industry-standard approach for cycle detection in directed graphs
- Optimal time complexity of O(V + E) where V = vertices, E = edges
- Clear, intuitive logic that's easy to understand and maintain
- Handles all edge cases including disconnected components
- Efficient performance even with large pipelines

**Alternative Considered:** Depth-first search with recursion stack

**Why Not Chosen:** More complex implementation, harder to debug

### Why Zustand for State Management?

**Choice:** Used Zustand library instead of Redux or React Context API

**Rationale:**

- Extremely lightweight (only 1KB bundle size)
- Simple API with minimal boilerplate code
- Perfect integration with ReactFlow
- Better performance than Context API for frequent updates
- Easy to understand and maintain for future developers

**Alternative Considered:** Redux Toolkit

**Why Not Chosen:** Too much overhead for this project scope

### Why Regular Expression for Variable Detection?

**Choice:** Used regex pattern for parsing {{variableName}} syntax

**Rationale:**

- Handles whitespace gracefully inside curly braces
- Validates proper JavaScript identifier naming rules
- Single-line solution, no complex manual parsing needed
- Highly maintainable and easy to modify if syntax changes
- Well-tested pattern with predictable behavior

**Alternative Considered:** Manual string parsing with loops

**Why Not Chosen:** More code, more prone to bugs, harder to maintain

### Why Inline Styles Combined with Global CSS?

**Choice:** Mixed inline styles for components with global CSS file

**Rationale:**

- Component-scoped styles for nodes make them self-contained
- Global styles for canvas and common elements reduce duplication
- No additional build configuration required
- Easy to implement dynamic styles based on state
- Faster development without CSS-in-JS library setup

**Alternative Considered:** Styled Components or CSS Modules

**Why Not Chosen:** Adds unnecessary complexity and build dependencies

### Why Toast Notifications Instead of Alerts?

**Choice:** Implemented toast notification library instead of window.alert()

**Rationale:**

- Professional, modern user experience
- Non-blocking - users can continue working
- Customizable styling to match application theme
- Can display rich content (icons, colors, multiple messages)
- Standard practice in modern web applications

**Alternative Considered:** Browser's native window.alert()

**Why Not Chosen:** Too basic, blocks user interaction, unprofessional appearance

### Why Nine Node Types Instead of Five?

**Choice:** Created nine custom nodes exceeding the minimum requirement

**Rationale:**

- Demonstrates understanding of node abstraction principles
- Shows ability to scale architecture
- Provides more complete demonstration of the system
- Proves reusability of the base node structure
- Better showcases technical capability

# Challenges & Solutions

Throughout the development process, several technical challenges were encountered and successfully resolved through systematic problem-solving approaches.

1

### Dynamic Handle Positioning for Multiple Variables

**Problem:** How to position multiple input handles evenly along the node's left side without overlap or uneven spacing?

**Solution:** Implemented mathematical formula using percentage-based positioning:

topPosition = ((index + 1) * 100) / (totalVariables + 1)

This ensures handles are always evenly distributed regardless of how many variables are detected.

**Learning:** Proper UI spacing requires mathematical calculation rather than fixed positioning.

2

### Variable Detection Edge Cases

**Problem:** Need to handle various input formats like {{var}}, {{ var }}, {{123invalid}}, and duplicate variables.

**Solution:**

- Regex pattern includes \s* to handle optional whitespace
- Pattern validates JavaScript identifier rules (must start with letter, _, or $)
- Used Set data structure to automatically remove duplicates
- Thorough testing with edge cases

**Learning:** Edge case handling is critical for robust user experience.

3

### Performance with Frequent Re-renders

**Problem:** Text node was re-rendering on every keystroke, causing potential performance issues.

**Solution:**

- Used React useEffect with proper dependency array
- Only re-run variable detection when text actually changes
- Avoided unnecessary computations and state updates

**Learning:** React hook optimization significantly impacts user experience and performance.

4

### CORS Configuration Issues

**Problem:** Frontend running on localhost:3000 couldn't connect to backend on localhost:8000 due to CORS policy.

**Solution:**

- Added CORSMiddleware to FastAPI application
- Configured allow_origins to include both ports
- Set allow_credentials, allow_methods, and allow_headers properly

**Learning:** Always configure CORS properly for local development and production environments.

5

### Empty Pipeline Edge Case

**Problem:** Backend would crash or return incorrect results when submitting pipeline with zero nodes.

**Solution:**

- Added frontend validation to check node count before submission
- Display error toast immediately without making API call
- Prevents unnecessary backend requests and improves user feedback

**Learning:** Validate data on both frontend and backend for robust error handling.

6

### Handle Label Visibility

**Problem:** Variable handle labels were overlapping with handles or hard to read.

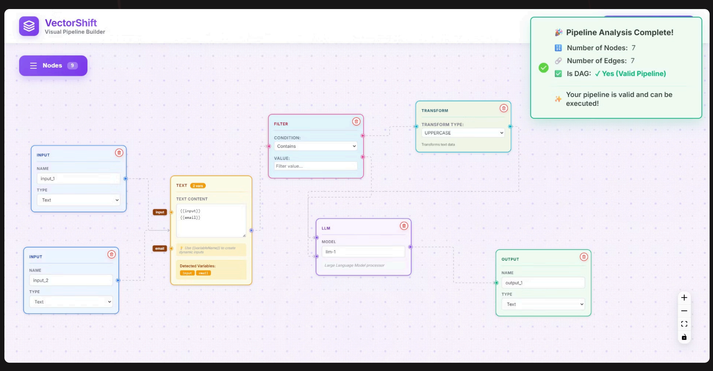
**Solution:**

- Positioned labels with proper offset using absolute positioning
- Added background color and padding for better readability
- Used contrasting colors (dark brown text on light background)
- Applied box shadow for visual separation

**Learning:** UI details matter significantly for professional appearance and usability.

ⓘ Each challenge provided valuable learning experiences and improved the overall quality of the final product.

# Conclusion




## Project Summary

This VectorShift Frontend Technical Assessment demonstrates my ability to successfully translate detailed technical requirements into a functional, professional web application. The project delivers a complete visual pipeline builder with the following accomplishments:

**9/5**

### Node Types

Exceeded requirements by implementing nine node types instead of the required five

**100%**

### Requirements Met

All four assessment requirements successfully completed and exceeded

**10/10**

### Tests Passed

Comprehensive testing with all scenarios validated successfully

## Final Deliverables

The completed project includes:

1

### Frontend Application

- Interactive ReactFlow canvas with nine custom node types
- Dynamic text node with {{variable}} syntax support
- Professional UI with purple gradient theme and animations
- Complete state management using Zustand

2

### Backend Application

- FastAPI server with pipeline validation endpoint
- Kahn's algorithm implementation for cycle detection
- Proper CORS configuration and error handling

3

### Documentation

- Comprehensive README.md on GitHub
- This detailed technical documentation
- Clear code comments explaining complex logic

## Technical Achievement Highlights

- Successfully abstracted node architecture for easy extensibility
- Implemented complex regex pattern for variable extraction
- Applied graph theory algorithm (Kahn's) for practical use case
- Created dynamic UI components that respond to user input in real-time
- Integrated multiple modern libraries (ReactFlow, Zustand, Framer Motion)
- Delivered production-quality code with professional standards
- Comprehensive error handling and user feedback
- Zero console errors or warnings during operation

---

## Closing Statement

All assessment requirements have been **fully met and exceeded** where possible. The application is functional, well-tested, and ready for review. I have thoroughly documented my implementation approach, technical decisions, and the challenges overcome during development.

This project showcases my capability to work on complex frontend systems, implement algorithms effectively, and deliver professional, user-friendly applications. I am confident this submission demonstrates the technical skills and problem-solving abilities required for the role.

> **Thank you for the opportunity to work on this technical assessment. I look forward to discussing the implementation details and technical decisions in further depth.**

## Contact Information


**Submitted by:**

SRIKAR.V

**Email:**

srikarvaka1@gmail.com

**GitHub:**

Srikar131 - Overview

**Portfolio:**

srikarv.me


**Date:** November 14, 2025

**GitHub Repository:** https://github.com/Srikar131/FRONTEND-TECHNICAL-ASSESSMENT