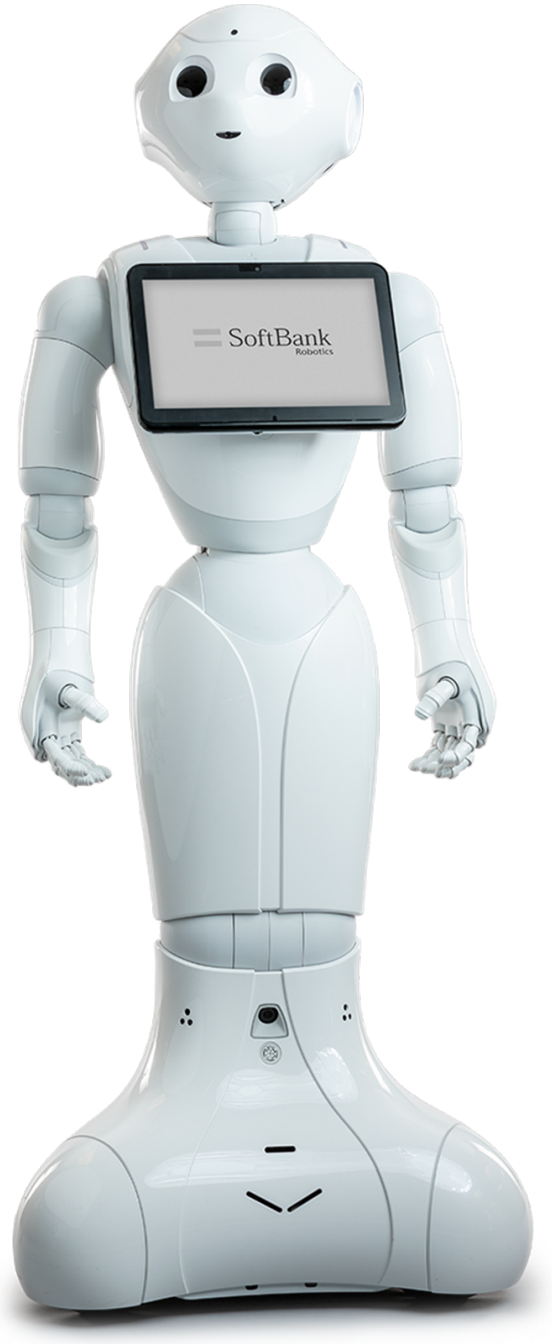
INFOSP - Fall 2025 - Developer Manual

Manual for Developers for Pepper+ Software

Pepper+ Team

January 2026



Contents

1 Introduction & Reading Guide

In this document, we provide potential future developers of Pepper$^+$ with the tools and knowledge they need to succeed in extending and improving our product. In Section 2, we provide a global overview of the architecture of the project, focusing heavily on the Control Backend. In Section 3, we go into detail about all components of the User Interface and provide concrete steps to change/add certain functionality. In Section 5, we go over the communication channels between the Robot Interface and the Control Backend and provide a checklist for implementing a new robot interface. We conclude by walking through some concrete examples of new functionality one might wish to add, explaining what code to change/add and what a developer can expect.

2 Architecture Overview

In this section, we discuss the global architecture of the project, as well as dive deeper into some aspects of the pipeline we use for decision making within the server backend.

2.1 Global Architecture

Our software consists of three components: the User Interface (UI), the Control Backend (CB), and the Robot Interface (RI). A high-level overview of these components and their interaction is shown in Figure 1. We will outline each of the components in this figure, as well as how they interact with each other, beginning with the UI.

User Interface The UI, which is a web application built using React/TypeScript, is the part of the software that an end user will interact with. It communicates to the CB using HTTP, either by listening to endpoints that send server-sent events for continuous streams of data (i.e., logs), or via REST API endpoints, exposed on both the UI and the CB. The UI enables a researcher to utilize all available features of the software in an intuitive and easy-to-navigate manner, acting as a shell around the core functionality in the CB. In the UI, a researcher can design and monitor a so-called behavior program. Such a program consists of one or more phases, each of which may contain a set of (conditional) norms, goals, and triggers. The UI is responsible for sending a behavior program to the CB, ensuring that it is correctly formatted for the CB to load. It consists of two main components: the program editor and the monitoring page.

The program editor is the dedicated tool for a researcher to define their programs and express their desired behavior in terms of nodes. The program editor provides a drag-and-drop menu to build a network of nodes, each node having its own function. The nodes have extra functionality to show their purpose and give feedback about the status and validity of their connections. Researchers also have the option to save and load their programs locally. When the 'Run Program' button, shown in Figure 2, is pressed, the program editor will compile all of the nodes and connections into a reduced program and send it to the control backend, starting the experiment. The Monitoring Page is then automatically loaded.

The Monitoring Page serves as a source of information and enables interaction during an active experiment. It provides information about the state of the current experiment, ranging from the current phase of the experiment to the conditions that are currently met. The page contains experiment controls to allow for managing the current phase and state of experiments. Additionally, it provides the researcher with speech and gesture controls to manually intervene during the experiment. To allow for more transparent monitoring, there is a built-in log window, which presents high-level information, like what the robot heard or said.

Control Backend The CB is the brain behind the software. It executes user-defined behavior programs, facilitates communication between the RI and the UI, manages LLM queries, and generates actions based on input. The CB is also responsible for the behavior and decision making of the robot. It does this by following the principles behind the belief-desire-intention (BDI) software model. The pipeline behind this will be covered in Section 2.2.1. To actually control the (physical) robot, it communicates with the RI,

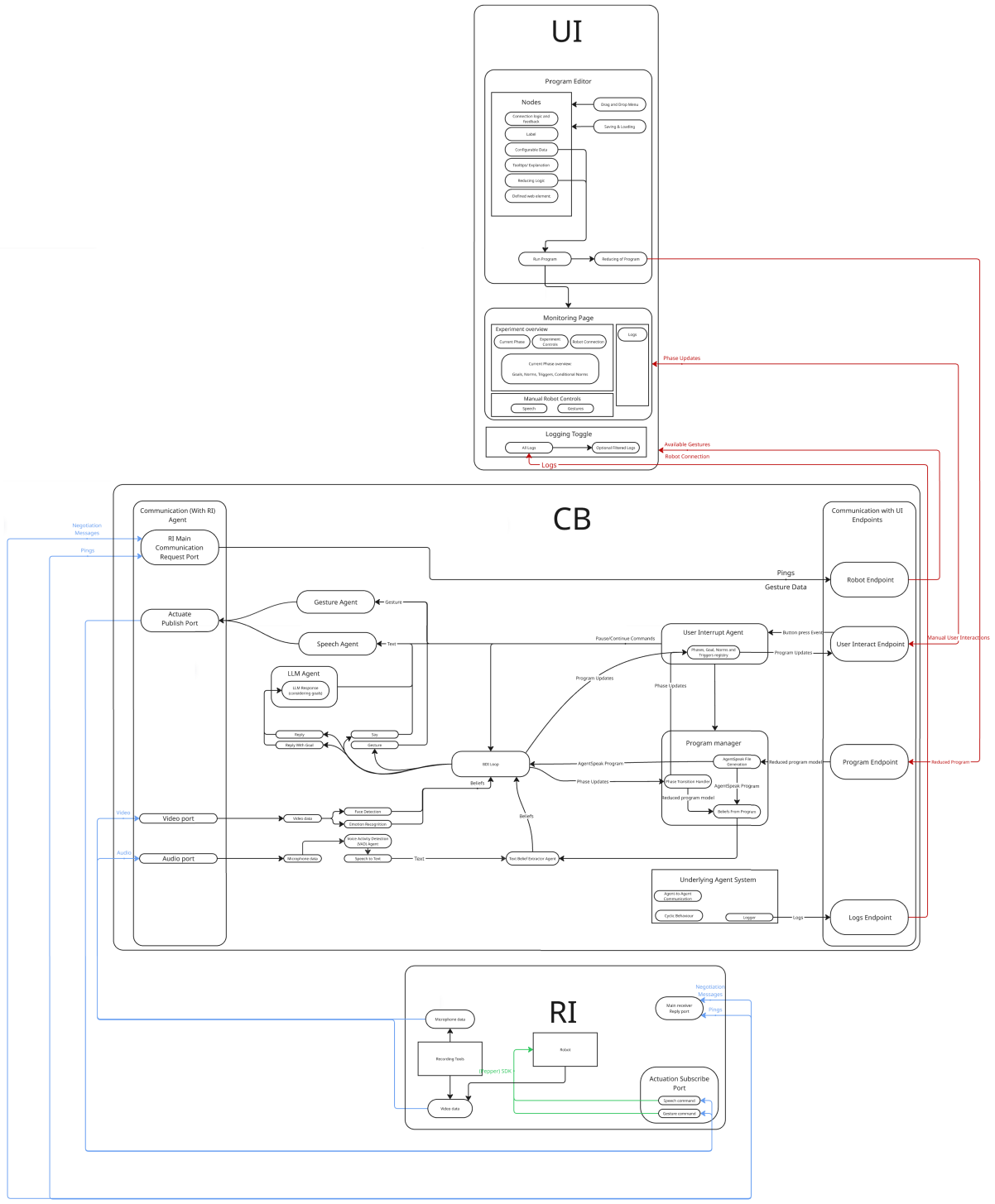


Figure 1: Overview of the software architecture with the three main components. Blue arrows represent connections over (ZeroMQ) sockets, while red arrows represent HTTP connections.

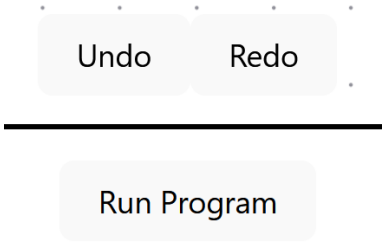


Figure 2: Undo, Redo, and Run Program buttons, located at the bottom of the visual programming page

which provides an API for executing certain behavior on the robot. Additionally, it informs the UI about the current state of the program, including the active phase, active norms, and the goal currently being pursued.

The Control Backend is a multi-agent system, with each agent having its own responsibilities. For example, some agents handle communication with the UI and RI, some are responsible for forming beliefs based on text/video, one is responsible for decision making, etc. By separating these agents into their own isolated instances, we make it easy to add and remove certain functionality, as well as to swap out certain agents in the future for ones with better performance. A detailed overview of the architecture of the CB is given in Section 2.2.

The CB is written in Python 3. As mentioned previously, it communicates with the UI through HTTP. RI communication is handled through sockets, specifically using the library ZeroMQ[**?**]. The considerations that went into choosing this library will be covered in Section **??**.

Robot interface The Robot Interface is the part of the software responsible for executing behavior on the robot, like saying a sentence or performing a gesture. The RI is an abstraction layer that ensures that we can, at a later point, introduce a different robot, which provides the same API in its RI, thereby eliminating the need to change any code in the CB. The RI provides an API to the CB through ZeroMQ sockets, through which it receives commands for the robot to execute. These commands include a specific endpoint, which can be used to distinguish functionality between commands. Implementation of how exactly these commands are translated to the robot is left to an individual interface.

An RI is directly linked to a single (physical or virtual) robot. Introducing support for a new robot only requires writing an RI for that robot (in whichever language you choose). In the case of the Pepper RI, it is written in Python 2.7.18, as that is the latest version of Python compatible with the Pepper software development kit (SDK).

2.2 CB Architecture

As the CB is the main component of our software, it is important to explain its architecture in detail. In this part, we first provide a global overview of how data flows through the CB (Section 2.2.1), after which we provide a detailed description of each agent in the CB (Section 2.2.2).

2.2.1 Data Flow

In this section, we outline the pipeline through which data flows, explaining at each point what happens to the data and why it might or might not be well-defined behavior. In Figure **??**, a more zoomed-in overview (compared to Figure 1) of the CB architecture is shown.

We have divided the data flow through the CB into three distinct phases: input processing and belief formation, intention formation, and action output. We will go into detail about each of these phases in the following parts.

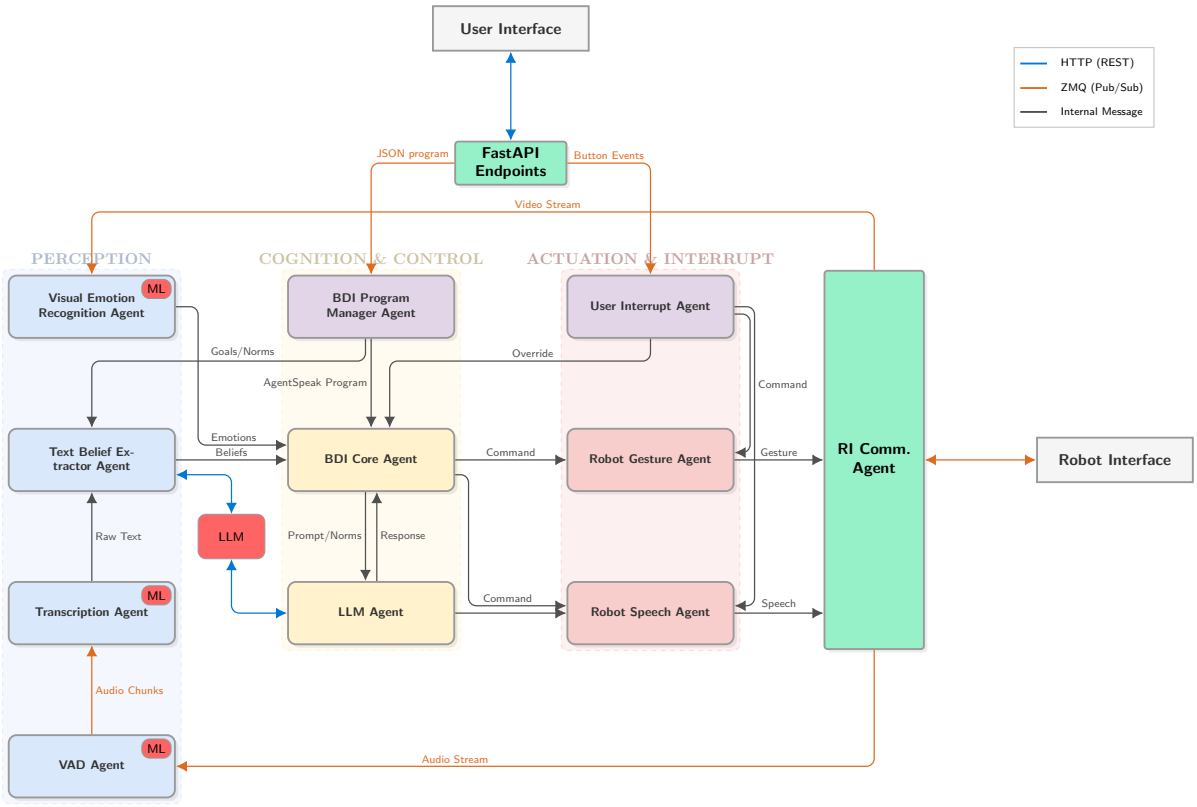


Figure 3: Overview of the agentic structure of the CB. Agents are grouped by their function and stage in the data flow pipeline. The darker red nodes in the top right corner of some agents represent pre-trained models whose function is encapsulated in the corresponding agent. Connections are color-coded, with blue connections indicating HTTP communication, orange indicating connection over sockets (specifically ZeroMQ sockets in this case) and gray indicating internal communication (simple string messsages between agents).

Phase 1: Input Processing and Belief Formation The CB receives audio and video data from the RI. The audio data is sent to the Voice Activity Detection Agent, which is responsible for detecting when a user is speaking. It then sends the speech fragments to the Transcription Agent, which converts the raw audio data to text. This text is sent to the Text Belief Extraction Agent. It is responsible for forming beliefs based on the transcribed text and the context of the conversation, and sending these beliefs to the BDI Core Agent. It also forwards the raw transcription to the BDI Core in the form of a `user_said("<text>")` belief.

The video data is sent to the Visual Emotion Recognition Agent, which is responsible for converting the raw video data into beliefs about what emotions are currently detected, and sending these beliefs to the BDI Core.

Since all of the agents in this phase use some form of AI (mostly pre-trained neural networks), the input-output relation is not strictly well-defined. Slightly different input might lead to differences in recognized emotions, text, etc. It is important to recognize this and treat the data received from these agents as partially incorrect. Balancing the minimization of model error with the maximization of performance is a delicate subject that requires a great deal of thought to be put into it, should the application require it.

Phase 2: Intention Formation The beliefs formed in the previous phase are sent to the "brain" of the CB: the BDI Core Agent. This agent is responsible for forming intentions (short-term goals). It does this by combining its desires (long-term goals and norms, specified by the researcher) with the beliefs formed in the

previous phase. The rules for combining beliefs and desires into intentions and actions are typically written in AgentSpeak [**?**]. This language, similar to prolog, allows us to control intention formation by defining plans that get executed when certain beliefs or goals are added to the belief base.

For our purposes, we convert the program created by the user in the UI into AgentSpeak code representing the same structure automatically. The BDI Program Manager handles this conversion through the use of a dedicated `AgentSpeakGenerator` class. As a short illustration, consider a very simple program that consists of a single phase with a single goal: find the user's name. This would result in the following core AgentSpeak plans:

```
phase(1).                          // we start off in the first (and only) phase

+user_said(Message)                // whenever the user says something, we execute this plan
  :   phase(1)                     // this plan is specifically for phase 1
  <-  !find_user_s_name.           // add the goal to find the user's name to our intentions

+!find_user_s_name                 // whenever we want to find the user's name, do this
  :   phase(1)
  <-  .say("What is your name?").  // call to a Python function to let the robot speak
```

Note that this file would not be functional in our program, as it omits several key elements in favor of brevity. This file gets parsed by the Program Manager, which in turn sends it to the BDI Core. The BDI Core uses the `agentspeak` library to abstract away some of the details of manually managing intentions and the BDI loop.

Phase 3: Action Output After the intentions are formed by the BDI Core Agent, they are converted into concrete actions (callable Python functions from within the AgentSpeak file). Inside these actions, the BDI Core forms gesture and speech commands, which it sends to the corresponding agents. the Robot Speech Agent and the Robot Gesture Agent are responsible for telling the robot what to do. These are relatively simple agents, interacting directly with the RI. They receive commands from the BDI Core, which they forward to the RI over ZeroMQ sockets.

2.2.2 Agent Descriptions

In this part, we discuss each agent mentioned previously in more detail. We outline their functionality and describe the mechanisms they use internally for achieving this functionality.

RI Communication Agent The RI Communication Agent is responsible for initializing communication with the RI, and checking whether the connection holds. It sends a negotiation message to the RI, which should respond with supported endpoints and the ports used for each. Using this response, the agent starts other agents, namely the VAD Agent, the Visual Emotion Recognition Agent, the Robot Speech Agent, and the Robot Gesture Agent. A continuous behavior of the RI Communication Agent is the periodic health check pings. If the RI is determined to be disconnected, this agent informs the UI and attempts re-negotiation. Note that this functionality is partially abstracted away in Figure 3. In actuality, the individual agents correspond directly with the RI, after connection is established through the RI Communication Agent.

VAD Agent The VAD Agent starts the Transcription Agent and listens continuously to the audio endpoint of the RI. It uses a pre-trained voice activity detection model to detect the beginning and end of a user's utterance, and when the end of an utterance is detected, it sends the a full audio fragment to the Transcription Agent.

Transcription Agent Receiving the audio fragments that the VAD Agent sends, the Transcription Agent uses a pre-trained transcription model to transcribe audio into text. It sends the transcription to the Text Belief Extractor Agent.

Text Belief Extractor Agent From conversation text, the Text Belief Extractor Agent uses an LLM to determine whether beliefs hold and whether goals have been achieved. When a belief changes or a goal is achieved, the Text Belief Extractor Agent sends a message to the BDI Core Agent.

The beliefs formed by this agent are two-fold. The first type of belief is the semantic belief. A semantic belief is a belief derived from the conversation history using an LLM. An example of such a belief could be: "The user is a pirate". Since this is determined by semantic meaning rather than exact keywords, the system can infer this belief from synonyms (e.g., "I am a privateer") or indirect expressions (e.g., "My name is Captain Jack Sparrow"), in addition to literal statements. The other type of belief that this agent is responsible for is the achievement of goals. During program creation, the user has specified a set of (sub)goals for each phase. Determining whether these goals are achieved (based on their name and description) is also the responsibility of this agent.

The way these beliefs are formed is through structured LLM calls. The agent has access to the conversation history and tasks an LLM with determining which beliefs (out of a set of available beliefs) hold. The Text Belief Extraction Agent gets the available beliefs and goals from the BDI Program Manager, which is another agent, responsible for keeping track of the currently loaded program.

Visual Emotion Recognition Agent The Visual Emotion Recognition Agent listens to the video endpoint of the RI, using a pre-trained emotion detection model to detect emotions in frames. In a window of time, it tracks the emotions of each face. At the end of the time window it determines the dominant emotion for each face and updates the BDI Core Agent about each individual emotion that is currently detected.

BDI Core Agent The BDI Core Agent runs the AgentSpeak program and uses inputs from most other agents as beliefs in this program. These beliefs are constantly updated as agents receive new information. The BDI Core receives the AgentSpeak program from the BDI Program Manager Agent, indirectly coming from the UI. Depending on which actions are triggered from within the AgentSpeak program, it might either directly give commands to the Robot Speech Agent and the Robot Gesture Agent, or first query the LLM Agent for a response. It then passes this response as speech commands to the Robot Speech Agent. Changes in state of the program, such as a phase change, which norms are active, and which goal is currently being chased, are shared with the User Interrupt Agent.

BDI Program Manager Agent The BDI Program Manager Agent receives programs from the UI and transforms them into AgentSpeak code. It resets agents that might keep track of which state the experiment is in, namely the LLM Agent and the Text Belief Extractor Agent. Then it shares the new program with the User Interrupt Agent, sends the AgentSpeak program to the BDI Core Agent, and shares information about which beliefs and goals should be tracked with the Text Belief Extractor Agent.

LLM Agent The LLM Agent is responsible for managing conversation history and generating responses using an LLM. It can receive prompts from the BDI Core, which include the current norms to follow and the message the user spoke. Optionally, a prompt can carry a temporary "goal", used to further steer the LLM in the desired direction. It queries the LLM for a response, which is directly streamed back to the BDI Core.

Robot Speech Agent The Robot Speech Agent receives `SpeechCommand`s from the BDI Core and the User Interrupt Agent, which it forwards directly to the RI over ZeroMQ sockets.

Robot Gesture Agent The Robot Gesture Agent receives `GestureCommand`s from the BDI Core and the User Interrupt Agent, which it forwards directly to the RI over ZeroMQ sockets.

User Interrupt Agent When a user performs a runtime interruption of a program, the User Interrupt Agent is responsible for forwarding this interruption to the BDI Core (or the actuation agents, if the interruption was a direct speech or gesture action for the robot to perform). It also receives updates about the currently-running program from the BDI Core, which it shares with the UI over a dedicated server-sent event endpoint.

3 User Interface

This section hopes to provide a detailed guide, as well as information about the workings of the User Interface (UI). We will provide extra context and explanation into all the components we use, as well as give a step-by-step guide on how to create a node from the ground up. First, we will talk a bit about how we have divided some of our code and assets, then we will go into the visual programming page and its implementation, following up we will provide information about the monitoring page, and finally we provide in which way we expect any connection and formats to the Control Backend (CB).

3.1 Separation of Assets, Components, Pages, and Utilities

In order to strive for more modularity, the User Interface is structured having separation of concerns in mind. All assets, meaning pictures or other external files, are stored in the `src/assets/` folder. Components that are accessed throughout the entire User Interface, are defined and stored in the first `src/components/` folder. Any other components that are directly involved in certain pages, or subpages, are defined in the corresponding `./components/` folder of that (sub)page.

3.2 Visual Programming Page

The following section will explain in which way the Programming Page is implemented, as well as inform about any places one can remove, add, or adjust the current features.

3.2.1 Basic ReactFlow implementation

The core of the Visual Programming Page is *ReactFlow*[1], in combination with *Zustand*[2]. ReactFlow is a customizable React component for building node-based editors and interactive diagrams. We define a FlowState type in `VisProgTypes.tsx` that defines the shape of the Zustand store used for managing the visual programming flow. This compound type includes defining the all with a type signature based on the ReactFlow, such as "OnConnect" and "OnNodesChange". We initialize our own FlowState based on this type in 'VisProgStores.tsx'. Within this file, we control the logic that is called through the interaction of the editor.

In case a developer would like to change the basic functionality at the core of how the visual programmer works, they can change the functions defined in our constant 'useFlowStore', within 'VisProgstores.tsx'. For example, if one had a function to celebrate a new node being added, they would only need to add a call to this function within the designated function 'addNode' contained inside our defined useFlowStore.

3.2.2 Visual Programming Components

For the Programming Page, we have specific components to help users create a complex diagram. Each of these components contain logic and visual elements for their defined purpose. In Table 1, we see a list of the types of components we have, with information about them.

The rest of this subsection will explain the components in more detail, going over their functionality, as well as how one can adjust them.

Drag and Drop Bar

Relevant files: DragDropSidebar.tsx

The drag and drop bar is a web element that provides the users with the ability to drag a node from the toolbar onto the existing program diagram, adding them to the editor. We define the complete element as a function 'DndToolbar()', which loops over all the default node data, and checks if the boolean 'droppable' is set to true. For each type of node for which this holds, this function creates calls 'DraggableNode()', a function that creates another web element within our toolbar. We pass a function to each node that, on

[1] https://reactflow.dev/
[2] https://zustand-demo.pmnd.rs/

Component	Main purpose of the component
Drag and Drop Bar	Allows users to drag and drop defined nodes
Warnings Bar	Notifies the user about any illegal or disfavored configurations
Rule-based Handles	Allow for complex configuration of connections between defined nodes
Save and Load Panel	Provides the ability to save and load created diagrams
Plan Creating	Serves as a medium for creating compound plans, needed for certain nodes

Table 1: Visual Programming Page Components and Functionality

drop, checks if the position is valid, and calls 'AddNodeToFlow()', correctly adding any nodes to our flow. This bar is designed with modularity in mind, meaning that defining a new node should automatically be correctly handled with the existing code. However, note that there is a specific CSS style called within this element, namely `.draggable-node-<node type>`, which should be defined in 'VisProg.module.css'.

Warnings Bar

Relevant files: EditorWarnings.tsx, WarningSidebar.tsx, WarningSidebar.css

The Warnings Bar serves as an extra precaution against the creation of unwanted programs. Inside 'EditorWarnings.tsx', all logic for the warning system is defined. It includes all types and severities, as well as a defined type for how the complete warning registry should be structured, alongside our used editor warning registry. This logic is used for our page element, defined in 'WarningSidebar.tsx', which consists of a header where you can change the severity filter, as well as the lists of all the warnings. The warnings are dynamically updated through the use of the program. Node-specific warnings are defined within their respective react elements, adjusting the registry where needed. The check for a complete program, and the warning arising from an incomplete program are defined in 'VisProg.tsx', and called on an update. The styles for the Warnings Bar are stored in 'WarningSidebar.css'.

In case a developer would like to adjust the types or severities of warnings available, they would have to adjust the type 'WarningType' and 'WarningSeverity' in EditorWarnings.tsx. Then, implementing this warning can be done from any point in the visual programming page, by (un)registering a warning, following the type of 'EditorWarning', using the FlowStores' 'registerWarning()' and 'unregisterWarning()' respectively.

Rule-based Handles

Relevant files: RuleBasedHandle.tsx, RuleBasedHandle.module.css, HandleRules.ts, HandleRuleLogic.ts

Rule-based handles are used to create a new layer of conditions for handles to adhere to before validating connection. This way any change of handle connection in the program editor is automatically enforced into a correct state. Definitions for a rule, as well as the registering and evaluation are defined in 'HandleRuleLogic.ts'. The 'HandleRule' typing is then used in 'HandleRules.ts' to define our own rules. In our case we have rules for allowing only connections from an array of types, allowing only connections from a certain handle, and disallowing connections to the node's own typing. These rules can be given arbitrarily when creating the handles, defined in 'RuleBasedHandle.tsx'. In 'RuleBasedHandle.tsx' there is a definition for a Single- and Multi-Connection Handle, which allow a single and multiple connection, respectively. These handles can be used within Nodes to give them their functionality. All the styles for the handle components are stored in 'RuleBasedHandle.module.css'.

In case a developer would like to add, or remove certain rules, they would have to add, or remove the rules in 'HandleRules.ts'. Next, they can use the rule when creating handles in their nodes. Rules are passed in an array, as an argument 'rules'. Note that the rules themself are defined to take another array as argument, so an example 'rules' variable **inside typescript** would be:

rules=[allowOnlyConnectionsFromHandle([nodeType:"phase",handleId:"data"])]

Save and Load Panel

Relevant files: SaveLoadPanel.tsx, SaveLoadPanel.module.css

The Save and Load Panel allows users to save their graph locally, and upload previously saved programs into the editor, recreating them. This also allows them to share programs via any means possible. It uses the 'SaveLoad.ts' file, defined in 'src\utils\' to create a Blob using the correct structure, and passes this object to create a URL to save this object, and passes it to the user. Currently, the Blob creation only takes nodes and edges into account, as well as a name for the program. If, in the future, a graph needs more information than the Nodes and Edges to (re)create a graph, this information should be encoded in this Blob as well. Note that this also means updating the 'HandleFileChange' to correctly adjust the Zustand store with the additional information. All the styles for the Save and Load Panel are contained in 'SaveLoadPanel.module.css'.

Plan Creating

Relevant files: Plan.tsx, Plan.default.ts, PlanEditor.tsx, PlanEditor.module.css, PlanEditingFunctions.tsx, GestureValueEditor.tsx, GestureValueEditor.module.css

The main component, the Plan Editor is a component that allows users to create a complex plan for certain nodes. A plan is collection of steps, like: 'say "hi"', or 'do gesture: "wave"'. A plan and its subparts are defined within 'Plan.tsx'. The default value for a plan is defined in 'Plan.default.ts'. The 'PlanEditor.tsx' contains the multipart component, starting with a button that opens a separate creation window. After opening the user is faced with two main parts; the left side, which is responsible for creating an action, and the right side, which shows all the steps of the current plan. Creating an action is subdivided into three options: Speech, Gesture, and LLM. All the creation logic for Speech and LLM action is contained within the Plan Editor. However, the Gesture Editor has its separate sub-component and logic, stored in 'GestureValueEditor.tsx'. It contains a toggle for switching between the "tag" mode and the "single" mode. Adding any of these actions to the plan using the "Add Step" button adds this action to the plan. Creating/ Saving the program updates the parent node with the updated plan. 'PlanEditingFunctions.tsx' contains two functions that take in a plan and insert, or delete a goal, respectively. When the program is run, these plan are reduced into useful data using the 'PlanReduce()' function within 'Plan.tsx'. The PlanEditor and GestureValueEditor component each have their own styles defined in their '.module.css' files.

For future development, it is essential that all the types of correct actions and their possible values are well-defined. Different types of actions can be redefined in 'Plan.tsx'. Note that changing this, in turn, means updating 'PlanEditor.tsx' to account for the change in types, needing a case for how their action should be build, as well as an interface with the user for input. The Gesture Value Editor component is responsible for fetching, or storing all of the correct types of gestures. In case this changes, this should be updated.

3.2.3 Visual Programming Nodes

At core, a node is a typing that represents everything ReactFlow needs to know about a node, extending the functionality of a Record. The basic definitions holds for our implementation, however, we expand upon this definition, by giving all of nodes additional responsibilities. A node is not a class, but rather a type that holds its own information compared to other nodes, and we add most of the functionality through the functions that defined in the same file as the typing. One function we have integrated into all of our nodes, and is essential to our visual programming page is the default export function, with the name of the node. It takes the respective node's properties as parameter, and returns a React element describing this node, allowing us to rending it in the graph. Next, we also have functions that are called in certain scenarios for all of our implemented nodes. One key function that is called is the 'Reduce' function. This function is responsible for reducing all the information a node holds into essential information for the Control Backend to receive, making sure that all the values are correctly parsed. Additionally, we have a default initialization of the information stored in a different file: '<Node>.default.ts'. All of our individual nodes are defined in 'nodes\'. The nodes we have currently defined are: the Basic Belief Node, the End Node, the Inferred Belief Node, the Norm Node, the Phase Node, the Start Node, and the Trigger Node.

3.2.4 Node registry

Next to the modular approach of having a specific file be responsible for the functionality of a node, another import factor is the Node Registry, which allows for polymorphism in code, reducing needed complexity in calling functions. The Node Registry serves a 'main hub', telling the rest of the program what functions or variables should be accessed for which types of nodes. These functions can then be called regardless of type in other parts, for example, when creating new nodes we can access the default data using its 'type':

const defaultData = NodeDefaults[type as keyof typeof NodeDefaults]

In this case, for example, we have defined all the default values of the nodes inside a variable 'NodeDefaults' inside 'NodeRegistry.ts'. The registry holds (references to) information about many different aspects, including the respective: React Component, Default values, Reducing algorithm, Connection Logic, Deletion Logic, Tooltips.

3.2.5 Creation of a new node

Now, we know the concise definition of our nodes, however, for future developers it is essential to understand the complete idea when creating new nodes. Therefor, we provide additional information, and a guide, to the creation of new nodes:

1. Create new files for the node

The first step will be to create two new files for the new node. **For this guide I will refer to the new node's type as '⟨newNodeType⟩'**. One of the two files created should be a file (preferably) named ⟨newNodeType⟩.tsx. This file will contain all the logic for the new node. The second file is a file (preferably) named ⟨newNodeType⟩.default.ts. This file will only contain the default values of the node.

2. Define the typing of the node

In your ⟨newNodeType⟩.tsx file, you need to define what kind of information your new type has. You can define this using

```
export type <newNodeType> Data = {
    // Any node-specific data goes here.
}
```

In this definition you can fill this typing with all the needed information. Note that, for our implementations of the Drag and Drop Bar and phases, your type should at least have parameters for: 'label' - a string that shows its label on the drag and drop bar, 'droppable' - a boolean that tells the drag and drop bar whether this node should be droppable or not Then, you should add

```
export type <newNodeType> = Node<<newNodeType>Data>'
 // For example: export type TestNode = Node<TestNodeData>
```

Adding this completes the initializing of our node as a React node. Note that this requires you to import the type 'Node' from '@xyflow/react'. Finally, make sure that this node has default values in the '.default.ts' file. This should be something like

```
export const <newNodeType>Defaults: <newNodeType>Data = {
    label: <Your label>,
    droppable: <True/False>,
    /*Make sure to add any other data of your new types signature here.*/
}
```

This marks the first definition of your new node! Let's implement all the needed functions to correctly show the new node in our graph, and use it.

3. Implementing the visuals

Now, we want to make sure that when we are able to add the new node to the graph, it correctly shows. In our software, we expect all the nodes to have their own default export function, that defines how the node should be rendered. You can add this like:

```
export default function <newNodeType>(props: NodeProps<<newNodeType>>) {
    /*Here will come the rendering logic*/
}
```

Now, you can already return any React.JSX.Element you would like to render, and this would work. However, with how our modular nodes are defined, we usually follow a structure like:

```
return (<>
{/*You can use a Toolbar here, to allow for tooltips and deletion.*/}
{/*import { Toolbar } from '../components/NodeComponents'; */}

// Make sure we use the correct styles
<div className={`${styles.defaultNode} ${styles.<YourStyle>}`}>
    <div className={"flex-row gap-sm"}>
      Your static text.
      {/* Feel free to use the 'TextField' function from our components for more
          dynamic text input */}
      {/* For textfields, you have to import TextField from '../../../../components/
          TextField'. */}
    </div>

    {/* Usually, we define the handle logic and creation here. */}
    {/* You can do this using the SingleConnectionHandle and MultiConnectionHandle
        */}
    {/* These are defined in '../HandleRules'.*/}
    {/* For now, here is a simple handle that allows multiple connections: */}
    <MultiConnectionHandle id="source" type="source" position={Position.Right}/>
</div>
</>)
```

Note that, you can use the properties of the node to allow for more dynamic visualization. Adding 'const data = props.data as ⟨newNodeType⟩Data;' before rendering allows you to more easily change the React Element based on dynamic preferences.

4. Adding our node to the registry

All of our defined nodes are linked together through the use of the registry. It allows other files to know the node exists, and use the needed functionality to correctly implement the node. Without adding the node to the registry, the program will 'never know' it exists.

Adding a node to the registry is quite simple. In `NodeRegistry.ts` we keep all the relevant functions and data together. In this file, you can see that we have defined variables which range from defining the different Node's types to their tooltips. For now, we should add our node to this registry, letting it know that we have this new type of node, with its own data, default values, and rendering function. Note that for the registry, we use a key that represents the **type** of your node, in this guide we indicate the exact key type as ⟨**newType**⟩.

You can follow the next idea, adding your functions and data to relevant variables:

```
export const NodeTypes = {
    start: StartNode,
    end: EndNode,
    // Other defined nodes like phase, norm, goal
    // You need to add this:
    <newType>: <newTypeNode>,
};

export const NodeDefaults = {
    start: StartNodeDefaults,
    end: EndNodeDefaults,
    // Other defined nodes standards, like phase, norm, goal
    // You need to add this:
    <newType>: <newTypeNode>Defaults,
};
```

Adding this basically means telling your program that your ⟨newType⟩ has the defaults and rendering function. You might notice there are a lot of different functions also being passed into the registry, from other nodes. These are also very important, and we will handle them in a later step. Note that every new global functionality to all the nodes, should have their own variable in the registry, mapping them the type keys, so we can easily look up all the nodes' functionality using their type.

Adding this already should allow you to see, drag and drop your new node unto the graph. Note that this new node does not have any connection logic yet, so connecting can result in console errors.

4. Adding connection logic

The next two important parts of a node are the connection logic functions, and the reducing logic. With every (dis)connection of nodes, a function gets called to both the source and target of the node, letting them know that a new connection was made. The editor expects all of the nodes to handle extra functionality themselves, so we need to add these functions, even if they are without additional logic:

```
export function <newType>ConnectionTarget(_thisNode: Node, _sourceNodeId: string) {
    // no additional connection logic exists yet
}

export function <newType>ConnectionSource(_thisNode: Node, _targetNodeId: string) {
    // no additional connection logic exists yet
}

export function <newType>DisconnectionTarget(_thisNode: Node, _sourceNodeId: string) {
    // no additional connection logic exists yet
}

export function <newType>DisconnectionSource(_thisNode: Node, _targetNodeId: string) {
    // no additional connection logic exists yet
}
```

Of course, new functionality means letting the registry know, so let us also add these functions on the correct places in the registry, in `NodeRegistry.tsx`.

```
// Connections
export const NodeConnections = {
  Targets: {
    start: StartConnectionTarget,
    end: EndConnectionTarget,
    // Other nodes, add this:
    <newType>: <newType>ConnectionTarget,
  },
  Sources: {
    start: StartConnectionSource,
    end: EndConnectionSource,
    // Other nodes, add this:
    <newType>: <newType>ConnectionSource,
  }
}

// Disconnections
export const NodeDisconnections = {
  Targets: {
    start: StartDisconnectionTarget,
    end: EndDisconnectionTarget,
    // Other nodes, add this:
    <newType>: <newType>DisconnectionTarget,
  },
  Sources: {
    start: StartDisconnectionSource,
    end: EndDisconnectionSource,
    // Other nodes, add this:
    <newType>: <newType>DisconnectionSource,
  },
}
```

These functions will them be called whenever there is a (dis)connection, accounting for whether the node was the source or target of the connection.

5. Adding reducing logic

In the case that this node should have data that is also send to the Control Backend, we should have a function that reduces the information of the node into the expected needed data. This is our reduce function. The reducer takes in the current node, and a possibly a list of nodes, and should return an output in the form of 'Record<string, unknown>', mapping the relevant data to the corresponding fields. A simple reducer could look something like:

```
export function <newType>Reduce(node: Node, _nodes: Node[]) {
  // You can use `const data = node.data as <newNodeType>Data;` for quicker access to
      data.
  return {
    id: node.id,
    reduceWorks: true,
    label: node.data.label,
  }
}
```

This will make sure that when the reduce function is called, we output the desired result. Again, we should add this to the registry (in `NodeRegistry.tsx`), to make sure it is called properly:

```
export const NodeReduces = {
  start: StartReduce,
  end: EndReduce,
  // Other nodes' reduce functions.
  // Add this:
  <newType>: <newType>Reduce,
}
```

Now, you should already be able to drag and drop the node, and connect and reduce properly. In case it is connected to a phase node, it will automatically be reduced into the '<newType>s:' field of the reduced phase, so for example a simple problem with the example reduce would looks like:

```
{
  "phases": [
    {
      "id": "<Generated UUID>",
      "name": "Phase 1",
      "norms": [],
      "goals": [],
      "triggers": [],
      "<newType>s": [
        {
          "id": "<Generated UUID>",
          "reduceWorks": true,
          "label": "<your label>"
        }
      ]
    }
  ]
}
```

6. Styling and CSS

You've now got a fully functional Node, however, it can look very bland. In order to have a more complete node, you should implement the correct styles and reference them. In case you added `${styles.<YourStyle>}` to your node's className, you should incorporate this in `VisProg.module.css`. Furthermore, when creating the drag and drop bar, we try to access the correct style using `styles.draggable-node-<newType>`. So for the basic style implementation you should add the following to `VisProg.module.css`:

```
.node-<newType> {
    /*Add a colour for the ouline and filter:*/
```

```
    outline: aquamarine solid 2pt;
    filter: drop-shadow(0 0 0.25rem aquamarine);
}

.draggable-node-<newType> {
    /* You can change this to your liking. */
    padding: 3px 10px;
    background-color: canvas;
    border-radius: 5pt;
    cursor: move;
    outline: aquamarine solid 2pt;
    filter: drop-shadow(0 0 0.75rem aquamarine);
}
```

Now you have a fully working and visual node. The only thing to do next, is to extend its functionality, and make sure that the program is even more happy!

7. Extra Functionality for Tooltips, Warnings and reducing

Our program would like all nodes to have their own tooltip, a small text which shows the users what you can do with a node. Just like all other functionality, this is done by defining itself in the node's file, and linking it with the registry:

```
// In <newNodeType>.tsx:
export const <newType>Tooltip =`
  This node is made for testing, this should show when the tooltip is shown.
  This happens when hovering on the node in the drag and drop bar, or
  when the tooltip is hovered on throughout the node's toolbar, in case it exists.`;

// In NodeRegistry.ts::
export const NodeTooltips = {
  start: StartTooltip,
  end: EndTooltip,
  //  Other nodes, add this:
  <newType>: <newType>Tooltip,
}
```

Next, we can add additional warnings to the system whenever something is not up to our likings. We do this using an useEffect() within the default <newNodeType> function. First, we need to define what warning we would give, according to the `EditorWarning` type defined in `EditorWarnings.tsx`, and then (un)register it accordingly:

```
// Inside of your default export <newNodeType> function:
{
    //...

    // Check for the warnings (a seperate function is more clean)
    check<newType>Warnings(props);

    // Return the React Element:
    // ...
}

// Then create a checking function:
function check<newType>Warnings(props: NodeProps<<NewNodeType>>) {
    // The functions should be retrieved from the flow store.
    const {registerWarning, unregisterWarning} = useFlowStore.getState();

    // Example: A custom warning for if this node is too far to the left.
    const tooFarLeftWarning : EditorWarning = {
        // Make sure to check the correct typings in '../components/EditorWarnings'.
          scope: {
            id: props.id,
            handleId: undefined,
          },
          type: 'CUSTOM_WARNING',
          severity: "INFO",
```

```
            description: "The <newNodeType> is a bit too far to the left of the graph!"
    };

    // Make sure to check the (un)registerWarnings' documentation!
    if (props.positionAbsoluteX < 30) { registerWarning(tooFarLeftWarning); }
    else { unregisterWarning(props.id, `${tooFarLeftWarning.type}`)}
}
```

Finally, we can also specify whether or not we want this node to be (automatically) reduced as a field of the Phase. By default, this is true, so the type automatically gets reduced in its own field. However, in case your node should not be reduced as a field of the phase, you can add the following in `NodeRegistry.ts`:

```
export const NodesInPhase = {
  start: () => false,
  end: () => false,
  // Other nodes:
  <newType>: () => false,
}
```

This tells the phase to NOT handle this node as a child, or field of itself when reducing.

Now that you have the grasp of creating a node in our system, your imagination is the limit! Make sure to always check for missing imports, and check the documentation whenever the functions seem hard to understand. An example file for a new node is provided in section **??** .

For implementing more functionality related to this node in the Monitoring Page, check section 3.3.5.

3.3 Monitoring Page

This section will break down the implementation of the features in the monitoring page as well as discuss the implementation of adding a new type of node in the Monitoring Page. Roughly the implementation of the Monitoring Page is able to be subdivided into two parts. The implementation of user interrupt events and the the implementation of state synchronization with the Control Backend.

3.3.1 Synchronization with Control Backend.

To maintain a real-time representation of the experiment, the frontend utilizes a custom hook, `useExperimentLogic`, which manages the local state for phase progression, active node IDs, and goal tracking.

State Management and Stream Listeners
Relevant files: MonitoringPageAPI.ts, MonitoringPage.tsx
The synchronization is driven by two Server-Sent Events (SSE) streams:

- **Experiment Stream**: Managed via `useExperimentLogger`, this stream handles `phase_update`, `goal_update`, and `trigger_update` events to shift the UI context as the robot progresses.
- **Status Stream**: Managed via `useStatusLogger`, this stream specifically listens for `cond_norms_state_update` to toggle the active status of conditional norms dynamically.

Initial Data Loading and Phase Transitions
Relevant files: MonitoringPage.tsx, MonitoringPageAPI.ts
When the experiment starts, the `MonitoringPage` relies on the `useProgramStore` to provide the full program structure. The initial state is established as follows:

- **Initial Phase**: The `useExperimentLogic` hook initializes `phaseIndex` to 0 and `activeIds` to an empty object.
- **PhaseDashboard**: This component acts as a filter. It uses getters from the store, such as `getGoalsWithDepth`, `getTriggersInPhase`, and `getNormsInPhase`, to extract only the nodes relevant to the currently active `phaseId`.

When a `phase_update` is received via the Experiment Stream, `handleStreamUpdate` searches for the index of the new `id` within the `phaseIds` array retrieved from the store. If found, `setPhaseIndex` updates the UI context, causing the `PhaseDashboard` to re-render with the goals and norms of the new phase, while resetting `goalIndex` to 0.

Experiment Completion
Relevant files: MonitoringPage.tsx
The transition to the final state is handled by a reserved ID check. If the incoming `phase_update` contains an ID of `"end"`, the `isFinished` state is set to `true`. In this state, the `PhaseProgressBar` marks all phases as completed, and the `PhaseDashboard` is replaced with a finished message.

3.3.2 User Interrupt Events

Here we will list the implementation of all the features that allow the user to intervene with the experiment, using buttons on the page. Generally speaking, when such button is pressed, relevant information is sent to the Control Backend via HTTP requests.

Send API Call
Relevant files: MonitoringPageAPI.ts
Most user interrupt events utilize a unified sender function called sendAPICall. This function expects a type and context. The type and context contain all the necessary information that the Control Backend needs for an interrupt event. Consider the following example calls:

```
1
2  //Make Pepper say specified sentence
3  sendAPICall("speech", "Hello, I'm Pepper")
4
5  //Make Pepper perform specified gesture
6  sendAPICall("gesture","animations/Stand/Emotions/Positive/Happy_1")
7
8  //Make Pepper override a goal/conditional norm/trigger with specified ID
9  sendAPICall("override_unachieve", String(item.id))
```

An endpoint could also be included if you want to send the information to a more specific endpoint. In our current implementation, this is not used and all the interrupt events are sent to the "*button_pressed*" endpoint.

Forced Speech
Relevant files: MonitoringPageComponents.tsx
Forced speech allows a developer to override the robot's current BDI (Belief-Desire-Intention) stack by inputting a direct utterance. This is implemented through two distinct React Functional Components that use the API call described in section 3.3.2:

The first component is the direct speech component. An input component that uses the React state via UseState to manage the text content. It supports both clicking events and the onKeyDown (Enter) event. When such event is triggered it sends an API call with type "speech" and the text content as its context. Afterwards it clears the text content (using UseState).

The second component contains a list of preset phrases. Each one of these contains a label, that is displayed on the page and a text property, which is the actual text that is passed through via the Send API Call (3.3.2). In our current implementation these two fields differ slightly to ensure that all text can be nicely displayed on the MonitoringPage without cluttering. To illustrate, consider this example:

```
1  {
2    label: "About yourself",
3    text: "Tell me something about yourself"
4  }
```

Currently, there are three presets, which are hard-coded in the component itself. As a developer you can add or edit any sentence that you want.

Forced Gesture
Relevant files: MonitoringPageComponents.tsx
Similar to the forced speech, one can also send gestures for Pepper to perform during the experiment. Right now a drop-down menu of a list of pre-selected gestures is presented to the user. The items in the list contain a label, which is presented on the Monitoring Page and a value, which is the hard-coded path of that specific gesture. It must be noted that this path is specific to Pepper. To illustrate, consider this example:

```
{
  label: "Wave",
  value: "animations/Stand/Gestures/Hey_1"
}
```

As a developer, you might want to swap these gestures to be compatible with a different Robot Interface. To do this, simply change the value to the specific path that contains the gesture on your specific gesture.

Design Note: A different approach to solving this problem would be to fetch the gestures available via your specific Robot Interface, although our software currently does not implement this.

The selected gesture is stored via React's UseState. An API call is sent (section 3.3.2) with the button's onClick event, containing the topic "gesture" and the path to the specific gesture as its context.

Overrides
Relevant files: MonitoringPageComponents.tsx
The `StatusList` allows manual state toggling for Goals, Triggers, and Conditional Norms. Currently, Goals and Triggers are only clickable when they are not achieved/active, whereas conditional norms is clickable both in active as in inactive state. The corresponding clickable icons (X for inactive/unachieved and ✓for active/achieved) trigger specific API calls:

- ✓ → **override**: Used to manually mark a goal as achieved or to activate a trigger or conditional norm.
- *X* → **override_unachieve**: Used to manually deactivate an active Conditional Norm.

For conditional norms and triggers, after an override is performed, the active status is not immediately updated in the UI, but rather by status updates sent by the CB as described in (section). For Goals, however, the frontend performs an optimistic update by calling `setActiveIds` immediately, providing instant visual feedback before the backend confirms the change via SSE. This is done, because letting the BDI core decide whether a goal is completed in our current implementation causes it to be de-synchronized.

3.3.3 Experiment Controls

Relevant files: MonitoringPage.tsx, MonitoringPageAPI.ts
The `ControlPanel` provides four main global actions:

- **Play/Pause**: Sends a "pause" API call with context "false" (to play) or "true" (to pause).
- **Next Phase**: Triggers `nextPhase()` to force the backend into the next defined phase.
- **Reset Experiment**: Triggers `resetExperiment`, which re-parses the visual program using `graphReducer`, clears local states, and restarts the backend logic via `runProgramm`.

3.3.4 Robot Connection Monitoring

Relevant files: MonitoringPageComponents.tsx
The `RobotConnected` component maintains a dedicated SSE connection to `/robot/ping_stream`. It listens for a boolean value indicating connection status and updates the UI with a "Robot is connected" or "Robot is disconnected" message accordingly.

3.3.5 Implementing your own node in MonitoringPage

If you have created a new type of node, you first need to consider if it is a node that you want updates on the monitoring page (e.g., its active/completion state) or if it is fine to not include it at all.

If visibility is required, you must register the node within the `MonitoringRegistry`. This involves mapping the node's internal state transitions—specifically those handled by the asynchronous `_process_bdi_message` logic—to the UI components. This ensures that when the agent is "querying" or "interrupted," the frontend reflects these states in real-time.

4 Control Backend

This section provides an in-depth guide for developers regarding the internal workings of the Control Backend (CB). While Section 2 explained the high-level responsibilities of the agents, this section focuses on the code implementation, extending the system, and the specific libraries used.

4.1 Core Infrastructure

The Control Backend is built upon a custom asynchronous agent framework. Understanding the base classes, configuration management, and communication layers is essential before creating new functionality.

4.1.1 Agent System

Relevant files: `src/control_backend/core/agent_system.py`, `src/control_backend/agents/base.py`

Every functional unit in the backend inherits from the `BaseAgent` class. This class provides the foundational infrastructure for lifecycle management, logging, and message passing.

When implementing a new agent, developers should inherit from `control_backend.agents.base.BaseAgent`. This specific base class ensures that the logger is automatically named correctly based on the package structure, whereas the core system base class does not.

Key methods available to every agent include:

- **setup()**: An abstract method that must be implemented. This is where sockets are connected and initial behaviors are started.
- **add_behavior(coroutine)**: Registers a long-running loop or a fire-and-forget task. The agent tracks these tasks and cancels them automatically upon shutdown.
- **send(message)**: Asynchronously sends an `InternalMessage` to another agent.
- **handle_message(message)**: An abstract method that acts as the handler for incoming messages.

4.1.2 Internal Messaging

Relevant files: `src/control_backend/schemas/internal_message.py`

Communication between agents is handled via the `InternalMessage` schema. This creates a uniform envelope for all data passed within the system.

Sending Messages
The system uses a hybrid routing approach. When you call `self.send()`, the system checks the destination:

1. **Direct Inbox:** If the destination agent is running in the same process (registered in the `AgentDirectory`), the message is placed directly into its asynchronous queue.
2. **ZeroMQ Fallback:** If the destination is not found locally, the message is serialized and broadcast over the internal ZeroMQ PUB/SUB bus.

Handling Messages
To react to messages sent by other agents, you must implement the `handle_message` method. The recommended pattern is to match against the `sender` and the `thread` (topic) of the message, and then use Pydantic to validate the body. It is recommended to implement new schemas for different message types in `src/control_backend/schemas/`. In the following example, we use the existing `BeliefMessage` schema as an illustration.

```
async def handle_message(self, msg: InternalMessage):
    # Filter by sender
    if msg.sender == settings.agent_settings.bdi_core_name:
        match msg.thread:
            case "beliefs":
                # Validate the JSON body into a Pydantic model
                try:
                    belief_msg = BeliefMessage.model_validate_json(msg.body)
                    self._process_beliefs(belief_msg)
                except ValidationError:
                    self.logger.error("Invalid belief message format.")
            case "shutdown":
                await self.stop()
            case _:
                self.logger.debug(f"Unhandled thread: {msg.thread}")
```

4.1.3 ZeroMQ Integration

Relevant files: `src/control_backend/core/config.py`

While 'InternalMessage' handles high-level agent coordination, raw ZeroMQ (ZMQ) sockets are often required for streaming data (like audio), connecting to external APIs (like the Robot Interface), or specialized low-latency channels. The system uses `zmq.asyncio` to integrate with the Python event loop.

Creating Sockets
Sockets should be initialized inside the `setup()` method of your agent. Always use the shared `Context.instance()` to ensure thread safety and proper context management.

```
import zmq
import zmq.asyncio as azmq

async def setup(self):
    context = azmq.Context.instance()

    # Example 1: Creating a Subscriber (SUB) socket
    self.sub_socket = context.socket(zmq.SUB)
    self.sub_socket.connect(settings.zmq_settings.internal_sub_address)
    self.sub_socket.subscribe(b"my_topic")

    # Example 2: Creating a Publisher (PUB) socket
    self.pub_socket = context.socket(zmq.PUB)
    self.pub_socket.bind("tcp://*:5555") # Bind allows others to connect

    # Start a behavior to listen to the socket
    self.add_behavior(self._listen_loop())
```

Using Sockets
To handle ZMQ messages without blocking the agent, use asynchronous reception within a behavior loop.

```
async def _listen_loop(self):
    while self._running:
        try:
            # wait for multipart message
            topic, body = await self.sub_socket.recv_multipart()
            # process data...
        except Exception:
            self.logger.exception("Error in ZMQ loop")
```

Cleanup
It is crucial to close sockets when the agent stops to prevent file descriptor leaks or port conflicts during restarts. Override the `stop()` method:

```
async def stop(self):
    if self.sub_socket:
        self.sub_socket.close()
    await super().stop()
```

4.1.4 Configuration

Relevant files: `src/control_backend/core/config.py`, `.env`

The system uses `pydantic-settings` for configuration management. All configurable parameters are defined in the `Settings` class and its nested sub-models (e.g., `AgentSettings`, `BehaviourSettings`).

To add a new configuration option:

1. Add the field to the relevant Pydantic model in `src/control_backend/core/config.py`.
2. (Optional) Add a default value directly in the class.
3. (Optional) Override the value using an environment variable in the `.env` file. Environment variables map to settings using double underscores (e.g., `behaviour_settings.sleep_s` becomes `BEHAVIOUR_SETTINGS__SLEEP_S`

4.2 Creating a New Agent

Extending the backend usually involves creating a new agent. The following guide outlines the standard procedure for implementing a new agent.

1. Create the Agent Class
Create a new file in the appropriate subdirectory of `src/control_backend/agents/`. Define a class that inherits from `BaseAgent`.

```
from control_backend.agents import BaseAgent
from control_backend.core.agent_system import InternalMessage

class ExampleAgent(BaseAgent):
    def __init__(self, name: str, **kwargs):
        super().__init__(name)
        # Initialize custom state here

    async def setup(self):
        self.logger.info("Setting up ExampleAgent")
        # Start background tasks
        self.add_behavior(self._custom_loop())

    async def handle_message(self, msg: InternalMessage):
        if msg.thread == "topic_name":
            self.logger.info(f"Received: {msg.body}")

    async def _custom_loop(self):
        while self._running:
            # Perform periodic logic
            await asyncio.sleep(1)
```

2. Register the Agent Name
Add a unique identifier for the agent in `src/control_backend/core/config.py` under the `AgentSettings` class. This ensures the name is centralized and configurable.

```
class AgentSettings(BaseModel):
    # ... existing agents ...
    example_agent_name: str = "example_agent"
```

3. Initialize in Main
Finally, instantiate and start the agent in the `lifespan` function within `src/control_backend/main.py`. Add the agent to the `agents_to_start` dictionary.

```
"ExampleAgent": (
    ExampleAgent,
    {
        "name": settings.agent_settings.example_agent_name,
    },
),
```

4.3 Extending the BDI System

Relevant files: `src/control_backend/agents/bdi/bdi_core_agent.py`, `src/control_backend/agents/bdi/default_behavior.asl`

The BDI (Belief-Desire-Intention) Core is the reasoning center of the robot. It runs AgentSpeak(L) code. A common task for developers is adding new "Actions"—Python functions that can be called directly from the AgentSpeak logic (e.g., to query an API, calculate a value, or trigger a complex hardware sequence).

4.3.1 Adding Custom Actions

Custom actions are defined in `BDICoreAgent._add_custom_actions`. The library `agentspeak` is used to bind Python functions to the AgentSpeak environment.

To add a new action, such as `.turn_around(Speed)`:

```
@self.actions.add(".turn_around", 1) # 1 is the number of arguments
def _turn_around(agent, term, intention):
    # 'term.args' contains the arguments passed from AgentSpeak
    speed = agentspeak.grounded(term.args[0], intention.scope)

    self.logger.info(f"Turning around at speed {speed}")

    # Perform the logic (e.g., send a message to a motor agent)
    # If the action is async, use self.add_behavior

    yield # Control must yield back to the BDI loop
```

Once registered, this can be used in the `.asl` file or generated program:

```
+!trigger_turn
  <- .turn_around(50).
```

4.3.2 AgentSpeak Generation and Pipeline Integration

Relevant files: `src/control_backend/schemas/program.py`, `src/control_backend/agents/bdi/agentspeak_generator.py`, `src/control_backend/agents/bdi/bdi_core_agent.py`

The backend receives behavioral programs from the UI in JSON format. Converting a new UI node into executable behavior is a multi-step pipeline process. If you introduce a new concept (e.g., a "Loop" node or a new type of "Condition"), you must update the system at several layers to ensure the data propagates correctly from the API to the BDI execution engine.

1. **Schema Definition** (`src/control_backend/schemas/program.py`):
   The entry point for the program is the Pydantic `Program` model. You must update this schema to accept the new node type. For example, if adding a new action type, you must define a Pydantic model for it (e.g., `LoopAction`) and register it in the `PlanElement` union type. This ensures the API accepts and validates the new JSON structure.

2. **AST Conversion** (`src/control_backend/agents/bdi/agentspeak_generator.py`):
   Once the data is validated, the `AgentSpeakGenerator` converts the Pydantic model into an Abstract Syntax Tree (AST). You must:
   - Update `agentspeak_ast.py` if new syntax structures are required.
   - Implement the `_astify` method for your new Pydantic model in the generator using the `@singledispatchmethod` decorator. This defines how your object translates to AgentSpeak string representation.
3. **Execution Logic** (`src/control_backend/agents/bdi/bdi_core_agent.py`):
   If your new element translates to a specific operation (e.g., `.wait_for_signal(X)`), the BDI agent needs to know how to execute it. You must register the corresponding Python function in `BDI Core` using `_add_custom_actions`.
4. **State Management** (`src/control_backend/agents/bdi/bdi_program_manager.py`):
   If the new element represents data that other agents need to know about (similar to how `Goals` and `Beliefs` are shared with the `TextBeliefExtractor`), you must update the `BDI Program Manager`. This agent is responsible for extracting static information from the program structure and routing it to the appropriate perception or monitoring agents before the AgentSpeak code is executed.

4.4 API and External Communication

Relevant files: `src/control_backend/api/v1/endpoints/`, `src/control_backend/api/v1/router.py`

The Control Backend exposes a REST API using FastAPI. This is primarily used by the UI to upload programs, trigger events, or retrieve logs.

4.4.1 Adding a New Endpoint

To add a new API route: 1. Create a new file in `src/control_backend/api/v1/endpoints/` (e.g., `custom.py`). 2. Define a router and the endpoint logic. 3. Access global state (like ZMQ sockets) via `request.app.state`.

```python
from fastapi import APIRouter, Request

router = APIRouter()

@router.post("/custom_trigger")
async def custom_endpoint(data: dict, request: Request):
    pub_socket = request.app.state.endpoints_pub_socket
    # Broadcast to internal agents
    await pub_socket.send_multipart([b"custom_topic", b"payload"])
    return {"status": "ok"}
```

4. Register the router in `src/control_backend/api/v1/router.py`.

4.5 Hardware Abstraction (Robot Interface)

Relevant files: `src/control_backend/schemas/ri_message.py`, `src/control_backend/agents/communication/ri_communication_agent.py`

The Control Backend communicates with the robot via the Robot Interface (RI) using ZeroMQ. The current implementation supports two primary actuation modes: **speech** and **gestures**.

Because the RI acts as a translation layer between our specific protocol and the robot's SDK (e.g., `qi` for Pepper), adding a new physical capability (such as specific head movements or LED control) requires a vertical update across the entire software stack. You cannot simply implement this in the Backend alone.

1. **Define the Protocol** (`src/control_backend/schemas/ri_message.py`):
   First, establish the contract between the CB and RI. Add a new `RIEndpoint` (e.g., `actuate/head`) and create a corresponding Pydantic model inheriting from `RIMessage` to validate the payload structure.

2. **Update the Control Backend**:
   You must create the logic to generate this command.
   - **BDI Core**: Register a new custom action (e.g., `.move_head(Yaw, Pitch)`) in `bdi_core_agent.py` so the behavior script can trigger it.
   - **Actuation Agent**: Update the `RobotGestureAgent` (or create a new `RobotMotorAgent`) to listen for this internal command and forward it to the RI via ZMQ.
3. **Update the Robot Interface**:
   The RI must be updated to listen on the new endpoint. You will need to write the specific handler that translates the incoming JSON payload into the robot-specific SDK calls.
4. **Update the User Interface**:
   If this new capability should be usable within a behavior program, you must update the UI's **Program Editor** to include a new Action Node type. This ensures the correct parameters are generated and sent to the CB during program upload.

5 Robot Interface

In this section, we will give an in-depth explanation about how the Robot Interface (RI) works, and how we expect future interfaces for robots to interact with the Control Backend (CB).

5.1 Our Robot Interface

The Robot Interface is an interface we have implemented with the goal of relaying useful information and commands from and to the *Pepper* [3] Robot. However, Pepper robot is quite outdated. Therefor we have designed the Robot Interface to be interchangeable, and adjustable for future developers. Here we will briefly explain how our current Robot Interface works.

Main Loop

Our current Robot Interface runs on Python 2.7, and uses *ZMQ* [4] sockets to communicate with the Control Backend. When we start our Robot Interface, we start up our communication sockets, and poll on the main thread until we receive a message from the Control Backend. Our different sockets have their own way of handling incoming communication, and/or handling outgoing communication.

Pepper communication

In order to complete the Robot Interface, we have functionality set up to interact with the robot, Pepper. This functionality depends on the *Qi* [5] library.

Default Interfaction

A default interaction with our Robot Interface follows a simple pattern. First, at the start of running the Robot Interface, the Robot Interface enters negotiation with the Control Backend. This is done using an endpoint `negotiate/ports`, within the Main Receiver. In this negotiation, the Robot Interface replies to a request send from the Control Backend. This request is on the endpoint, without any additional data, where the reply answers given the data of the ports. The data replied is formatted as a list, containing `id`, `port`, and `bind` data, with the option for additional data for specific endpoints.

5.2 Future Robot Interfaces

Due to our Robot Inteface being designed to be replacable, we want to give delopers a clear idea about how the robot interface should be changed to ensure proper functionality. For giving a clear idea, in the rest of this subsection, we will explain workings with the idea of creating a **new interface**.

[3] https://us.softbankrobotics.com/pepper
[4] https://zeromq.org/languages/python/
[5] http://doc.aldebaran.com/libqi/

Core Functionality

First, we need to make sure that the Robot Interface is compatible with the Control Backend, meaning that the our program does not crash. The first step in doing this is ensuring that the new interface uses ZMQ sockets to negotiate with the Control Backend. Our main receiver port is defined in the `src\robot_interface\core\config.py` file. This current set to port `5555`. Note that this same port is also defined in the Control Backend, meaning that a change of the main port should be reflected in both repositories' settings/ config file. Since the Control Backend initiates a request port (`zmq.REQ`), the Robot Interface should initiate a reply port (`zmq.REP`) on the same port.

Next, the new interface should be able to reply to any request, making sure that the communication loops keeps working correctly. The main port is accessed for two things: negotiation, and pings. The new reply port should handle both of these. When the reply port receives a request on endpoint `ping`, it should simply respond on the same endpoint with the same data. However, the more important part is ensure correct negotiation. Negotation involves updating our Control Backend about the correct ports of the individual endpoints. In our case, we have five different endpoints which can be negotiated: the main receiver, actuation, video, audio, and face recognition. Note that, for the core functionality, it is not **required** to negotiate all these ports, however, you would lose their functionality completely, of course. This can also result in code blocking within individual agents of the Control Backend.

For a better implementation of a new interface, the interface should implement all of the different endpoints. Here is an overview of the different endpoints:

Actuation
The actuation endpoint is the a receiver that handles commands for the robot. It received commands by subscribing (`zmq.SUB`) on the `actuation` endpoint, which is defined in the `config` file, differentiating between three different kinds: `actuation/speech`, `actuation/gesture/tag`, and `actuation/gesture/single`. The new Robot Interface should handle all the speech commands that are received in this endpoint, next to all the different gestures that are send through either `actuation/gesture/tag` and `actuation/gesture/tag`. This means that any new interface that works with a different robot should include logic for communicating to the robot, or handling invalid request gracefully.

Audio
The audio endpoint is a sender that sends audio to the Control Backend. It uses a publication (`zmq.PUB`) socket on the defined port in the `config` file. The current audio is sent as raw 32-bit float PCM data, in chunks of 512 samples and a sample rate of 16 kHz. Sent as bytes, not JSON. Make sure that the newly sent audio is also sent in a compatible manner, by using the same format.

Video
The video endpoint is a sender that sends video to the Control Backend. It uses a publication (`zmq.PUB`) socket on the defined port in the `config` file. The current video is sent as 24-bit BGR 640x480px, at 15 FPS. Sent as bytes, not JSON. Make sure that the newly sent video is also sent in a compatible manner, by using the same format.

Face
The face endpoint is a communication endpoint that replies with face recognition data to the Control Backend. It uses a reply (`zmq.REP`) socket on the defined port in `config`. The Control Backend sends a message on the endpoint without data, and after receiving this endpoint will send a boolean as reply, indicating whether or not a face is currently detected. Make sure that in the new interface, this format is also followed.

New Interface Checklist

In order to be more certain about the validity of a new interface, we give a checklist about the requirements for a complete new interface:

- The Robot Interface is constantly running.
- The Robot Interface is not blocked by a single endpoint receiver or sender.

- The Robot Interface implements a main receiver, on the main port, which is able to negotiate about the existing endpoints in the correct format, as well as reply to pings.
- The Robot Interface implements an audio endpoint, which sends audio in the correct format. This Endpoint is negotiated in the negotiation process.
- The Robot Interface implements a video endpoint, which sends video in the correct format. This Endpoint is negotiated in the negotiation process.
- The Robot Interface implements an actuation endpoint, which relays any speech and gestures command to the robot. This endpoint is negotiated in the negotiation process.
- The Robot Interface implements a Face endpoint, which sends the face detection status on request.