Module Interface Specification for Software Engineering

Team #23, Project Proxi
Savinay Chhabra
Amanbeer Singh Minhas
Gourob Podder
Ajay Singh Grewal

January 28, 2026

1 Revision History

Date	Version	Notes
2025-11-02	1.0	Initial draft created. Included module hierarchy and core Proxi design overview.
2025-11-03	1.1	Added detailed MIS for HH-IO, HH-Auto, and BH-Input modules.
2025-11-04	1.2	Completed BH-NLU, BH-Plan, BH-Safety, BH-Feedback, and BH-UI sections with semantics and tables.
2025-11-07	1.3	Added refined symbols and abbreviations and reflection section
2025-11-11	1.4	Final formatting, line-width cleanup, and rubric alignment for submission.
2025-01-21	1.4	Rev 0, implemented TA Feedback

2 Symbols, Abbreviations and Acronyms

See SRS documentation at https://github.com/gkpodder/Capstone/blob/design/docs/SRS-Volere/SRS.pdf

Additional symbols, abbreviations, and acronyms specific to this document are listed below.

Symbol / Term	Definition
:=	Assignment operator used for state transitions.
$\geq, \leq$	Greater than or equal to, less than or equal to.
$\rightarrow$	Function mapping or transition arrow.
Action	Atomic executable system behaviour.
BH	Behaviour-Hiding module (e.g., BH-Input, BH-Plan).
ExecStatus	Execution result (Pending, Success, or Failed).
FUNC.R.#	Functional requirement number from the SRS.
Hazard ID	Identifier from Hazard Analysis (e.g., H1, H2).
HH	Hardware-Hiding module (e.g., HH-IO, HH-Auto).
InputMode	Input type (VoiceOnly, TextOnly, Mixed).
Intent	Structured interpretation of user command text.
MCP	Modular Command Protocol agent interface.
OutputMode	Output type (VoiceOnly, TextOnly, Both).
Plan	Structured action sequence from BH-Plan.
QA	Quality Assurance (software testing process).
RiskLevel	Safety classification for user actions.
SD	Software-Decision module (e.g., SD-Types, SD-Log).
SRS	System Requirements Specification.
STT	Speech-to-Text (audio input converted to text).
TTS	Text-to-Speech (text output converted to speech).
UI	User Interface.
V&V	Verification and Validation Plan.

Contents

3 Introduction

This document presents the Module Interface Specification (MIS) for Proxi, an intelligent voice first assistant developed as part of the SFWRENG 4G06A Capstone Design Project at McMaster University. Proxi enables users to interact with their computer using speech or text commands and executes actions through a modular command protocol (MCP) agent system. The goal is to improve accessibility and user productivity by providing hands free, context aware computer control.

The MIS defines the interface and behaviour of each module described in the Module Guide. Each module encapsulates a single responsibility and follows the principles of information hiding and low coupling. Together these modules define how Proxi captures user input, interprets intent, plans actions, confirms operations, and provides feedback.

Complementary documents include:

- System Requirements Specification (SRS): https://github.com/gkpodder/Capstone/blob/design/docs/SRS-Volere/SRS.pdf
- Verification and Validation Plan (V&V): https://github.com/gkpodder/Capstone/blob/design/docs/VnVPlan/VnVPlan.pdf
- Hazard Analysis Report: https://github.com/gkpodder/Capstone/blob/design/docs/HazardAnalysis/HazardAnalysis.pdf

All documents are stored in the team's public repository at https://github.com/gkpodder/Capstone/. The SRS and V&V Plan define the measurable requirements and testing criteria that this MIS traces to. The Hazard Analysis identifies potential risks, such as unintended system actions, which are addressed by the BH-Safety module.
This MIS uses the structure and conventions from Hoffman and Strooper (1995) and Ghezzi et al. (2003). Each module is presented with clear syntax, semantics, and local function definitions, ensuring traceability between requirements, design, and testing. The notation style for states, transitions, and functions follows the mathematical conventions outlined in Section 4. Overall, this document provides a formal yet readable description of Proxi's modular design, supporting future implementation, testing, and maintenance.

4 Notation

The structure of the MIS for modules comes from Hoffman and Strooper (1995), with the addition that template modules have been adapted from Ghezzi et al. (2003). The mathematical notation comes from Chapter 3 of Hoffman and Strooper (1995). For instance, the symbol := is used for a multiple assignment statement and conditional rules follow the form $(c_1 \Rightarrow r_1 | c_2 \Rightarrow r_2 | ... | c_n \Rightarrow r_n)$.
The following table summarizes the basic and derived data types used by Proxi modules.

Data Type	Description
char	A single character or digit.
integer	Whole number in the range $(-\infty, \infty)$.
real	Number with fractional part, used for timing or duration.
boolean	Logical value in {true, false}.
string	Sequence of characters.
sequence(T)	Ordered list of elements of type T.
tuple(T_1,T_2, ...)	Finite ordered group of typed values.
AudioStream	Representation of sampled voice input.
Intent	Structured record containing type and parameters.
Plan	Record defining tool, parameters, and execution time.
Action	Atomic operation triggered by a plan or intent.
RiskLevel	Enum {Low, Medium, High} for safety checks.
ExecStatus	Enum {Pending, Success, Failed} for task outcomes.

The specification of Software Engineering uses some derived data types: sequences, strings, and tuples. Sequences are lists filled with elements of the same data type. Strings are sequences of characters. Tuples contain a list of values, potentially of different types. In addition, Software Engineering uses functions, which are defined by the data types of their inputs and outputs.
Derived functions and transitions are written in the form $f : A \to B$ to indicate a function mapping from type A to type B. Local functions such as *nextState* or *policy* are used to describe the intended mathematical behaviour of modules but may not exist as explicit code implementations.

5 Module Decomposition

The Proxi is decomposed into a hierarchy of modules following the design principles of information hiding and separation of concerns. Each module corresponds to a well-defined secret and is independently assignable to a developer. The decomposition balances hardware hiding, behaviour hiding, and software decision modules.

Table 1: Module Hierarchy for Proxi Voice Assistant

Level 1	Level 2 Modules (Secrets / Responsibilities)
Hardware-Hiding	**HH-IO (Audio I/O Adapter)** manages microphone input and audio output across different platforms.
	HH-Auto (Desktop Automation) performs basic desktop actions such as typing, clicking, or launching applications.
Behaviour-Hiding	**BH-Input (Voice & Text Manager)** captures user input, converts speech to text, and normalizes text commands. Implements *FUNC.R.1–R.2*.
	BH-NLU (Intent Parser) interprets text into structured intents and parameters based on defined command patterns. Implements *FUNC.R.3*.
	BH-Plan (Task Executor) determines which agent or tool should handle a command and coordinates its execution. Implements *FUNC.R.4*.
	BH-Safety (Confirmation Gate) validates actions that may affect files or system settings and requests confirmation. Implements *FUNC.R.9*.
	BH-Session (Context Manager) maintains user session data, history, and undo information for continuity. Supports *FUNC.R.4*.
	BH-Feedback (Response Manager) converts textual responses into spoken or visual feedback for the user. Implements *FUNC.R.5–R.6*.
	BH-UI (Proxi Interface) presents status updates, confirmations, and results; supports full voice-only interaction. Implements *FUNC.R.8*.
Software-Decision	**SD-Types (Core Structures)** defines abstract data types for Command, Intent, and Plan.
	SD-ToolRegistry (Action Map) maintains the mapping between recognized intents and available system actions.
	SD-Store (Local Storage) handles persistent storage for user preferences, session history, and logs.
	SD-AIClient (AI Service Client) provides configuration and communication for STT and text synthesis services.
	SD-Log (Event Logger) records system actions and feedback events for debugging and validation.

Likely Changes:

- The choice of speech recognition or text-to-speech library (for example, switching from Whisper API to a local model).
- Adjustments to the user interface layout or how voice commands trigger visible feedback or audio playback.
- Fine-tuning thresholds for speech detection and timing between input and response based on user testing.
- Updating supported voice commands or adding new MCP tools as features are expanded.

Unlikely Changes:

- The main processing loop of Input $\rightarrow$ Interpret $\rightarrow$ Plan $\rightarrow$ Execute $\rightarrow$ Feedback.
- The core data structures used for storing Commands, Intents, and Action Plans.
- A local, reliable, request-response communication style between modules via the MCP agent interface.

Traceability to SRS:

- **BH-Input** fulfills *FUNC.R.1–R.2*: speech and text input handling with accuracy $\geq 90\%$.
- **BH-NLU** fulfills *FUNC.R.3*: intent recognition accuracy $\geq 90\%$.
- **BH-Plan** fulfills *FUNC.R.4*: agent planning and execution success rate $\geq 85\%$.
- **BH-Feedback** fulfills *FUNC.R.5–R.6*: provides feedback and spoken confirmation within response time $\leq 2\,\text{s}$.
- **BH-UI** fulfills *FUNC.R.8*: supports full hands-free operation for accessibility.
- **BH-Safety** fulfills *FUNC.R.9*: requests confirmation before executing high-risk or destructive actions.
- **Support modules (SD, HH)** enable non-functional goals on latency, reliability, and auditability through structured logging.

6 MIS of HH-IO (Audio I/O Adapter)

6.1 Module

HH-IO (Audio I/O Adapter)

6.2 Type

Abstract object.

6.3 Uses

System audio interface

6.4 Syntax

6.4.1 Exported Constants

None.

6.4.2 Exported Access Programs

Name	Input	Output	Errors
initAudio	N/A	N/A	AudioInitFailed
openMic	N/A	N/A	MicNotFound
closeMic	N/A	N/A	CloseFailed
recordAudio	seconds: real	sound: AudioStream	RecordFailed
playAudio	sound: AudioStream	N/A	PlaybackFailed

6.5 Semantics

6.5.1 State Variables

- micOpen : boolean

6.5.2 Environment Variables

- micDevice : physical microphone
- speakerDevice : physical speaker or headset

6.5.3 Assumptions

- At least one working microphone and speaker device exists.
- Only one module controls the microphone at a time.
- **initAudio** is called before any other access routine.

6.5.4 Access Routine Semantics

initAudio():

- transition: micOpen := false if audio devices exist.
- exception: AudioInitFailed if audio devices cannot be initialized.

openMic():

- transition: micOpen := true if micDevice is available.
- exception: MicNotFound if micDevice is missing or busy.

closeMic():

- transition: micOpen := false if open.
- exception: CloseFailed if device cannot close.

recordAudio(seconds):

- output: returns an AudioStream captured from micDevice for *seconds*.
- exception: RecordFailed if capture fails or micOpen = $false$.

playAudio(sound):

- transition: speakerDevice plays *sound*.
- exception: PlaybackFailed if playback fails.

6.5.5 Local Functions

None.

7 MIS of HH-Auto (Desktop Automation)

7.1 Module

HH-Auto (Desktop Automation)

7.2 Type

Abstract object.

7.3 Uses

Operating system automation interface

7.4 Syntax

7.4.1 Exported Constants

None.

7.4.2 Exported Access Programs

Name	Input	Output	Errors
initAuto	N/A	N/A	AutoInitFailed
moveCursor	p: ScreenPos	N/A	ActionError
leftClick	N/A	N/A	ActionError
typeText	t: String	N/A	ActionError
openApp	id: AppId	N/A	ActionError

7.5 Semantics

7.5.1 State Variables

None.

7.5.2 Environment Variables

- desktopEnv : user desktop environment
- keyboardDevice : keyboard input channel
- pointingDevice : mouse or trackpad input channel
- pointerPos : current cursor position on the active display

7.5.3 Assumptions

- The user session allows simulated input events.
- Screen coordinates are valid for the active display.
- AppId refers to an installed and accessible application.
- **initAuto** is called before any other access routine.

7.5.4 Access Routine Semantics

initAuto():

- transition: automation channels to desktopEnv, keyboardDevice, and pointingDevice are established.
- exception: AutoInitFailed if automation cannot be initialized.

moveCursor(p):

- transition: pointerPos := p.
- exception: ActionError if cursor move fails.

leftClick():

- transition: a left-click event is emitted on pointingDevice at pointerPos.
- exception: ActionError if click event fails.

typeText(t):

- transition: keystroke events for t are emitted on keyboardDevice.
- exception: ActionError if key input fails.

openApp(id):

- transition: desktopEnv launches the application identified by id.
- exception: ActionError if app launch fails.

7.5.5 Local Functions

None.

8 MIS of BH-Input (Voice & Text Manager)

8.1 Module

BH-Input (Voice & Text Manager)

8.2 Type

Abstract object.

8.3 Uses

HH-IO, SD-AIClient, SD-Types, SD-Log

8.4 Syntax

8.4.1 Exported Constants

None.

8.4.2 Exported Access Programs

Name	Input	Output	Errors
initInput	N/A	N/A	InputInitFailed
startCapture	mode: InputMode	N/A	MicUnavailable, AlreadyCapturing
stopCapture	N/A	N/A	NotCapturing
getLastText	N/A	text: String	NoInputAvailable
getStatus	N/A	s: InputStatus	N/A

8.5 Semantics

8.5.1 State Variables

- inputState : InputState
- currentMode : InputMode
- lastText : String
- partialText : String
- lastError : InputError or null

InputState = {Idle, Listening, Processing}
InputMode = {VoiceOnly, TextOnly, Mixed}
InputStatus is a record:

- state : InputState
- hasText : boolean
- hasError : boolean

8.5.2 Environment Variables

- micStream : handled by HH-IO for live audio
- sttService : speech-to-text service client (through SD-AIClient)
- now : system clock for timing

8.5.3 Assumptions

- The microphone and STT component are available when started.
- Only one capture session runs at a time.
- Calling modules handle all exceptions raised.
- **initInput** is called before any other access routine.

8.5.4 Access Routine Semantics

initInput():

- transition:

$$\text{inputState}, \text{currentMode}, \text{lastText}, \text{partialText} := \text{Idle}, \text{TextOnly}, \text{""}, \text{""}$$

- exception: InputInitFailed if dependencies cannot be initialized.

startCapture(mode):

- transition: if inputState = Idle and micStream ready then

$$\text{inputState}, \text{currentMode}, \text{partialText} := \text{Listening}, \text{mode}, \text{""}$$

- exception: MicUnavailable if device fails, AlreadyCapturing if inputState $\neq$ Idle.

stopCapture():

- transition: if inputState $\neq$ Idle then inputState := Idle.
- exception: NotCapturing if inputState = Idle.

getLastText():

- output: returns lastText if not empty.
- exception: NoInputAvailable if lastText is empty.

getStatus():

- output: returns a record s with s.state = inputState, s.hasText = (lastText $\neq$ ""), s.hasError = (lastError $\neq$ *null*).

8.5.5 Local Functions

Let Event = {StartCmd, StopCmd, Chunk, Error, Timeout}
nextState : InputState × Event → InputState

$$\text{nextState}(s, e) = \begin{cases} \text{Listening} & \text{if } s = \text{Idle} \wedge e = \text{StartCmd} \\ \text{Idle} & \text{if } s = \text{Listening} \wedge e = \text{StopCmd} \\ \text{Processing} & \text{if } s = \text{Listening} \wedge e = \text{Chunk} \\ \text{Processing} & \text{if } s = \text{Processing} \wedge e = \text{Chunk} \\ \text{Idle} & \text{if } e = \text{Error} \vee e = \text{Timeout} \\ s & \text{otherwise} \end{cases}$$

During execution BH-Input updates

$$\text{inputState} := \text{nextState}(\text{inputState}, e)$$

for each event e. When transcription ends, partialText moves into lastText.

9 MIS of BH-NLU (Intent Parser)

9.1 Module

BH-NLU (Intent Parser)

9.2 Type

Library.

9.3 Uses

SD-Types, SD-Log

9.4 Syntax

9.4.1 Exported Access Programs

Name	Input	Output	Errors
parseText	text: String	i: Intent	ParseError

9.5 Semantics

9.5.1 State Variables

None.

9.5.2 Environment Variables

None.

9.5.3 Assumptions

- Input text may be noisy, incomplete, or ungrammatical.
- Command patterns and parameter schemas are defined in SD-Types.

9.5.4 Access Routine Semantics

parseText(text):

- output: produces an Intent record with fields

$$i.\text{type} = \text{detectCommand}(\text{text}), \quad i.\text{params} = \text{extractParams}(\text{text})$$

- exception: ParseError if text cannot be matched to any pattern.

9.5.5 Local Functions

$$\text{detectCommand} : \text{String} \to \text{IntentType}$$

$$\text{extractParams} : \text{String} \to \text{ParamSet}$$

detectCommand matches the normalized input against a finite set of command patterns (keywords, regular expressions, and templates) defined in SD-Types. extractParams parses any parameters required by the selected pattern (for example, filenames, application identifiers, or search text) using the parameter schemas from SD-Types. Both routines are deterministic and do not require external network calls.

10 MIS of BH-Plan (Task Executor)

10.1 Module

BH-Plan (Task Executor)

10.2 Type

Abstract object.

10.3 Uses

BH-NLU, SD-ToolRegistry, SD-Types, SD-Log, HH-Auto

10.4 Syntax

10.4.1 Exported Constants

ExecStatus = {Pending, Success, Failed}

10.4.2 Exported Access Programs

Name	Input	Output	Errors
initPlan	N/A	N/A	PlanInitFailed
planAction	i: Intent	p: Plan	NoToolFound
executePlan	p: Plan	s: ExecStatus	ExecError
cancelPlan	N/A	N/A	NoPendingPlan
getLastStatus	N/A	s: ExecStatus	N/A

10.5 Semantics

10.5.1 State Variables

- currentPlan : Plan or null
- lastStatus : ExecStatus

10.5.2 Environment Variables

- toolSet : accessible system tools or MCP agents
- now : system clock for execution timing

10.5.3 Assumptions

- The input intent has been validated by BH-NLU.
- Each available tool in SD-ToolRegistry exposes a run() routine.
- MCP agents or automation tools are reachable when requested.
- **initPlan** is called before any other access routine.

10.5.4 Access Routine Semantics

initPlan():

- transition:

$$\text{currentPlan}, \text{lastStatus} := \mathit{null}, \text{Pending}$$

- exception: PlanInitFailed if tool registry cannot be accessed.

planAction(i):

- output: generates a Plan record p with:

$$p.\text{tool} = \text{matchTool}(i.\text{type}), \quad p.\text{parameters} = i.\text{params}, \quad p.\text{time} = \text{now}$$

- transition: currentPlan := p.
- exception: NoToolFound if matchTool fails.

executePlan(p):

- transition:

$$\text{currentPlan}, \text{lastStatus} := p, \text{runTool}(p.\text{tool}, p.\text{parameters})$$

- output: returns lastStatus.
- exception: ExecError if runTool fails unexpectedly.

cancelPlan():

- transition: currentPlan, lastStatus := $null$, Failed.
- exception: NoPendingPlan if currentPlan = null.

getLastStatus():

- output: returns lastStatus.

10.5.5 Planning Logic

Define:

$$\text{matchTool} : \text{IntentType} \to \text{ToolId}$$

$$\text{runTool} : \text{ToolId} \times \text{ParamSet} \to \text{ExecStatus}$$

The planning decision can be expressed as:

$$\text{planAction}(i) = \begin{cases} p = (\text{matchTool}(i.\text{type}), i.\text{params}, \text{now}) & \text{if a tool exists for } i.\text{type} \\ \text{NoToolFound error} & \text{otherwise} \end{cases}$$

Execution behaviour follows:

$$executePlan(p) = \begin{cases} Success & \text{if runTool(p.tool, p.parameters) = true} \\ Failed & \text{otherwise} \end{cases}$$

10.5.6 Local Functions

- **matchTool**(t): searches SD-ToolRegistry for a matching tool.
- **runTool**(id, params): calls the linked MCP or system command.

11 MIS of BH-Safety (Confirmation Gate)

11.1 Module

BH-Safety (Confirmation Gate)

11.2 Type

Abstract object.

11.3 Uses

BH-UI, SD-Types, SD-Log

11.4 Syntax

11.4.1 Exported Constants

RiskLevel = {Low, Medium, High}
SafetyDecision = {AutoAllow, AskUser, Block}
ApprovalResult = {Approved, Denied, Cancelled}

11.4.2 Exported Access Programs

Name	Input	Output	Errors
classifyAction	a: Action	r: RiskLevel	N/A
decidePolicy	a: Action	d: SafetyDecision	N/A
confirmAction	a: Action	res: ApprovalResult	UserTimeout

11.5 Semantics

11.5.1 State Variables

- pendingAction : Action or null
- lastDecision : SafetyDecision or null

11.5.2 Environment Variables

- uiChannel : connection to BH-UI for user prompts
- now : system clock for time limits

11.5.3 Assumptions

- BH-UI can show a yes/no prompt and return a user response.
- Every Action record includes a defined riskLevel field.
- The system clock is monotonic for timeout checks.

11.5.4 Access Routine Semantics

classifyAction(a):
Output:

$$r := a.\text{riskLevel}$$

Transition: none.

decidePolicy(a):

- output: returns d of type SafetyDecision, where

$$d = \begin{cases} AutoAllow & \text{if } a.riskLevel = Low \\ AskUser & \text{if } a.riskLevel = Medium \\ Block & \text{if } a.riskLevel = High \wedge a.isIrreversible \\ AskUser & \text{if } a.riskLevel = High \wedge \neg a.isIrreversible \end{cases}$$

- transition: lastDecision := d.

confirmAction(a):

- transition: pendingAction := a.
- output:
  - If decidePolicy(a) = AutoAllow then res = Approved.
  - If decidePolicy(a) = Block then res = Denied.
  - If decidePolicy(a) = AskUser then BH-UI prompts user.
- exception: UserTimeout if no answer before time limit.

11.5.5 Risk Policy Table

Risk	Irreversible?	Decision	Example
Low	N/A	AutoAllow	Open folder, read file
Medium	N/A	AskUser	Rename or move file
High	false	AskUser	Delete to recycle bin
High	true	Block	Permanently delete data

11.5.6 Local Functions

$$\text{policy} : \text{RiskLevel} \times \text{boolean} \to \text{SafetyDecision}$$

$$\text{policy}(r, irr) = \begin{cases} \text{AutoAllow} & \text{if } r = \text{Low} \\ \text{AskUser} & \text{if } r = \text{Medium} \\ \text{Block} & \text{if } r = \text{High} \land irr = \text{true} \\ \text{AskUser} & \text{if } r = \text{High} \land irr = \text{false} \end{cases}$$

This module fulfills *FUNC.R.9* by ensuring confirmation or blocking of high-risk actions, reducing hazards identified in the safety analysis.

12 MIS of BH-Feedback (Response Manager)

12.1 Module

BH-Feedback (Response Manager)

12.2 Type

Abstract object.

12.3 Uses

HH-IO, SD-AIClient, SD-Types, SD-Log

12.4 Syntax

12.4.1 Exported Constants

OutputMode = {VoiceOnly, TextOnly, Both}

FeedbackStatus = {Idle, Speaking, Completed, Failed}

12.4.2 Exported Access Programs

Name	Input	Output	Errors
initFeedback	N/A	N/A	FeedbackInitFailed
queueMessage	msg: String, m: OutputMode	N/A	QueueFull
speakNow	msg: String, m: OutputMode	s: FeedbackStatus	TtsError
getLastStatus	N/A	s: FeedbackStatus	N/A
cancelAll	N/A	N/A	N/A

12.5 Semantics

12.5.1 State Variables

- outputQueue : sequence of (String, OutputMode)
- lastStatus : FeedbackStatus
- isSpeaking : boolean

12.5.2 Environment Variables

- audioOut : speaker connection through HH-IO
- ttsService : text-to-speech component (through SD-AIClient)
- uiChannel : text output channel to BH-UI

12.5.3 Assumptions

- ttsService can turn any short message into speech in less than the required response time from the SRS.
- HH-IO can play audio without blocking the whole system.
- The output queue has a fixed maximum size.
- **initFeedback** is called before any other access routine.

12.5.4 Access Routine Semantics

initFeedback():

- transition:

$$\text{outputQueue}, \text{lastStatus}, \text{isSpeaking} := [], \text{Idle}, \text{false}$$

- exception: FeedbackInitFailed if dependencies cannot be initialized.

queueMessage(msg, m):

- transition: if the queue is not full then append (msg, m) to outputQueue.
- exception: QueueFull if appending would exceed the limit.

speakNow(msg, m):

- transition: $\text{isSpeaking}, \text{lastStatus} := \text{true}, \text{Speaking}$.
- transition: if $m = \text{VoiceOnly}$ or $m = \text{Both}$ then ttsService produces speech for msg and HH-IO plays it on audioOut.
- transition: if $m = \text{TextOnly}$ or $m = \text{Both}$ then uiChannel displays msg.
- output:
  - If both requested outputs succeed, then $\text{isSpeaking}, \text{lastStatus} := \text{false}, \text{Completed}$.
  - Otherwise $\text{isSpeaking}, \text{lastStatus} := \text{false}, \text{Failed}$.
- exception: TtsError if ttsService cannot produce speech.

getLastStatus():

- output: returns lastStatus.

cancelAll():

- transition:

$$\text{outputQueue}, \text{isSpeaking}, \text{lastStatus} := [], \text{false}, \text{Idle}$$

12.5.5 Local Functions

We model the processing of the queue with a helper function:

$$\text{nextMessage} : \text{sequence of (String, OutputMode)} \to (\text{String}, \text{OutputMode}) \cup \{\text{None}\}$$

$$\text{nextMessage}(q) = \begin{cases} \text{first element of } q & \text{if } q \neq [] \\ \text{None} & \text{if } q = [] \end{cases}$$

BH-Feedback periodically checks outputQueue. If nextMessage returns a pair (msg, m), it behaves as in speakNow(msg, m) and then removes that entry from the queue. If None, it leaves the state unchanged.
This module fulfills *FUNC.R.5–R.6* by providing timely spoken and visual feedback to the user, and by reporting a clear status that can be logged or shown in the interface.

13 MIS of BH-UI (Proxi Interface)

13.1 Module

BH-UI (Proxi Interface)

13.2 Type

Abstract object.

13.3 Uses

BH-Feedback, BH-Safety, BH-Input, SD-Log, SD-Types

13.4 Syntax

13.4.1 Exported Constants

UIState = {Idle, Listening, Waiting, Displaying, Error}

13.4.2 Exported Access Programs

Name	Input	Output	Errors
initUI	N/A	N/A	UiInitFailed
updateView	s: UIState	N/A	RenderError
showMessage	msg: String	N/A	RenderError
promptUser	q: String	ans: Bool	Timeout
showStatus	st: FeedbackStatus	N/A	N/A
clearScreen	N/A	N/A	N/A

13.5 Semantics

13.5.1 State Variables

- uiState : UIState
- lastMsg : String
- lastError : String or null

13.5.2 Environment Variables

- display : visual interface (screen or console)
- micLED : visual cue showing listening status
- inputChannel : keyboard or voice response channel for prompts

13.5.3 Assumptions

- Display device is available and writable.
- Voice cues or LEDs can toggle quickly without delay.
- Text is short enough to fit within screen limits.
- **initUI** is called before any other access routine.

13.5.4 Access Routine Semantics

initUI():

- transition:
$$\text{uiState}, \text{lastMsg}, \text{lastError} := \text{Idle}, \text{""}, null$$
- exception: UiInitFailed if display cannot be initialized.

updateView(s):

- transition: uiState := s and display updates status text and micLED according to s.
- exception: RenderError if update fails.

showMessage(msg):

- transition: $\text{lastMsg}, \text{uiState} := msg, \text{Displaying}$.
- transition: display renders msg (and BH-Feedback may speak it, depending on OutputMode chosen by the caller).

- exception: RenderError if display fails.

promptUser(q):

- transition: uiState := Waiting and display renders q.
- output: returns ans as a yes/no value from inputChannel.
- exception: Timeout if no input received in time limit.

showStatus(st):

- transition: display shows latest FeedbackStatus st.

clearScreen():

- transition: uiState, lastMsg := Idle, "" and display clears the visual area.

13.5.5 Local Functions

We define a helper that maps states to display text:

$$\text{stateText} : \text{UIState} \to \text{String}$$

$$\text{stateText}(s) = \begin{cases} \text{“Listening...”} & \text{if } s = \text{Listening} \\ \text{“Waiting for input...”} & \text{if } s = \text{Waiting} \\ \text{“Processing...”} & \text{if } s = \text{Displaying} \\ \text{“Idle”} & \text{if } s = \text{Idle} \\ \text{“Error”} & \text{if } s = \text{Error} \end{cases}$$

This mapping helps BH-Feedback and BH-Safety show consistent notifications through both visual and voice channels. This module fulfills *FUNC.R.8* by ensuring full hands-free interaction and clear accessibility feedback for all system states.

References

Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.

Daniel M. Hoffman and Paul A. Strooper. *Software Design, Automated Testing, and Maintenance: A Practical Approach*. International Thomson Computer Press, New York, NY, USA, 1995. URL http://citeseer.ist.psu.edu/428727.html.

14 Appendix

[Extra information if required —SS]

Appendix — Reflection

[Not required for CAS 741 projects —SS]

The information in this section will be used to evaluate the team members on the graduate attribute of Problem Analysis and Design.

The purpose of reflection questions is to give you a chance to assess your own learning and that of your group as a whole, and to find ways to improve in the future. Reflection is an important part of the learning process. Reflection is also an essential component of a successful software development process.

Reflections are most interesting and useful when they're honest, even if the stories they tell are imperfect. You will be marked based on your depth of thought and analysis, and not based on the content of the reflections themselves. Thus, for full marks we encourage you to answer openly and honestly and to avoid simply writing "what you think the evaluator wants to hear."

Please answer the following questions. Some questions can be answered on the team level, but where appropriate, each team member should write their own response:

1. What went well while writing this deliverable?

2. What pain points did you experience during this deliverable, and how did you resolve them?

3. Which of your design decisions stemmed from speaking to your client(s) or a proxy (e.g. your peers, stakeholders, potential users)? For those that were not, why, and where did they come from?

4. While creating the design doc, what parts of your other documents (e.g. requirements, hazard analysis, etc), it any, needed to be changed, and why?

5. What are the limitations of your solution? Put another way, given unlimited resources, what could you do to make the project better? (LO_ProbSolutions)

6. Give a brief overview of other design solutions you considered. What are the benefits and tradeoffs of those other designs compared with the chosen design? From all the potential options, why did you select the documented design? (LO_Explores)

Amanbeer Minhas Reflection

1. **What went well while writing this deliverable?**

   Once I had the SRS, VnV Plan, and Hazard Analysis in place, this design doc felt much more manageable. Breaking the system into BH-Input, BH-Plan, BH-Safety, BH-Feedback, and BH-UI made it easier to see how everything fit together. Courses like COMPSCI/SFWRENG 3RA3 (Requirements) helped me think in terms of clear

responsibilities and traceable requirements. My QA co-op experience also helped because I naturally thought about how each module would be tested while I was specifying it.

2. **What pain points did you experience during this deliverable, and how did you resolve them?**

   The main pain point was finding the right level of abstraction. At first, I wrote the MIS in terms of specific tools and APIs, which made the design feel tied to one implementation. After revisiting the lecture notes and examples, I rewrote the modules to focus on behaviour and information hiding instead. Another challenge was formal notation for state and transitions. I drew on ideas from SFWRENG 2DM3 (Discret Maths) to think in terms of states, events, and transitions, then simplified that down to the most important pieces so the document would still be readable.

3. **Which of your design decisions stemmed from speaking to your client(s) or a proxy (e.g. peers, stakeholders, users)?**

   Most of our client feedback came from our team mates parents/relatives trying early voice-based prototypes. Some people wanted fewer confirmations so the system felt faster, while others were worried about accidental destructive actions. This made us directly include the BH-Safety module: we introduced risk levels and different behaviours for low, medium, and high risk actions. Ideas around accessibility and clear feedback were reinforced by discussions in ENGINEER 4A03, where we talked about inclusivity, ethics, and duty of care. Where we did not have direct user input, I leaned on course examples and the capstone lectures for reference.

4. **While creating the design doc, what parts of your other documents (e.g., requirements, hazard analysis, etc), if any, needed to be changed, and why?**

   While writing the MIS, I noticed that some requirements in the SRS were too vague to map cleanly to modules. We will refine a few of them to have clearer success conditions (for example, accuracy targets and timing bounds), so they matched the behaviour of BH-Input and BH-Feedback. The Hazard Analysis also needs change: some risks originally assigned to "the system" were moved specifically to BH-Safety, since that is where confirmation and blocking actually happen. The VnV Plan needs change to add module-level tests for voice accuracy, plan execution success, and safety prompts, reflecting what I learned about test design in my QA co-op and SFWRENG 3S03 (Software testing).

5. **What are the limitations of your solution? Put another way, given unlimited resources, what could you do to make the project better? (LO_ProbSolutions)**

   Right now, the design assumes relatively simple commands and a limited amount of context. The NLU is closer to pattern matching than full natural language understanding. The system also depends on reliable speech models and may not perform as well in noisy environments or for all accents. With more time and resources, we

would explore more robust language models, better noise handling, and wider accessibility testing with real users. We would also add more automation around logging and replaying user sessions for regression testing, drawing from ideas in my QA work and SFWRENG 3S03 and STATS 3Y03 for analyzing failure patterns.

6. **Give a brief overview of other design solutions you considered. What are the benefits and tradeoffs of those other designs compared with the chosen design? From all the potential options, why did you select the documented design? (LO_Explores)**

   We considered three main designs. The first was a single monolithic voice assistant where input, planning, and safety were all handled in one loop. This would have been easier to code quickly but very hard to test or change without breaking things. The second option relied mostly on an external cloud assistant, which might give better speech accuracy but raises privacy and reliability concerns. The third, which we chose, is the modular design in this document, with separate BH modules and clear interfaces. It fits the ideas from SFWRENG 2OP3, 2C03, 3RA3, and 3XB3 about modularity, testing, and requirements traceability. We chose it because it balances what we can realistically implement as a student with the level of structure and safety we have learned to aim for in software engineering.

Ajay Singh Grewal Reflection

1. **What went well while writing this deliverable?**

   What went well was that we were really able to define clear modules and map them to the requirements. This made it straightforward to write the MIS for each module.

2. **What pain points did you experience during this deliverable, and how did you resolve them?**

   A main pain point was having everyone be consistent with their ideas of how to write the MIS. We resolved this by having team meetings to discuss the format and style we wanted to use, and then sharing examples to ensure everyone was on the same page. Also, breaking down the system into smaller modules helped make this task more easy to handle.

3. **Which of your design decisions stemmed from speaking to your client(s) or a proxy (e.g. peers, stakeholders, users)?**

   From discussions with our peers and stakeholders, we chose a voice first design with simple visual feedback to ensure accessibility. We also decided to include a safety module to confirm high risk actions, as some users expressed concerns with poteintial mistakes.

4. **While creating the design doc, what parts of your other documents (e.g., requirements, hazard analysis, etc), if any, needed to be changed, and why?**

   While creating the design doc, we realized that some requirements in the SRS were not descriptive enough to be effectively mapped to moduleds. We roughly altered these requirements so they are more specific and measurable.

5. **What are the limitations of your solution? Put another way, given unlimited resources, what could you do to make the project better? (LO_ProbSolutions)**

   The limitations of our solution is that it is designed for a single desktop enviroment and heavlily is reliant on STT and LLM. Hence, offline use cases and strong privacy guarantees are not well supported. Given unlimited resources, we would explore more defined language models, and richer accessibility features to target a broader user base.

6. **Give a brief overview of other design solutions you considered. What are the benefits and tradeoffs of those other designs compared with the chosen design? From all the potential options, why did you select the documented design? (LO_Explores)**

   We had also considered a command line interface design and browser extension. However, these designs were not as user friendly and accessible as a voice first design. Hence, we chose the current documented design as it balances usability, accessibility, and modularity.

Savinay Chhabra Reflection

1. **What went well while writing this deliverable?**

   This deliverable was a lot smoother than the previous deliverables. We finished a good portion of the work before the TA meeting which allowed us to get good feedback on the deliverable before submission. We did a much better job at dividing the work this time and were more punctual in our delivery.

2. **What pain points did you experience during this deliverable, and how did you resolve them?**

   The pain point during this deliverable was being able to define the design without being too solution oriented. It is very easily to use specific services as we have done a good amount of work on the POC. This implementation might change for the final project so it's important not to pick any specific implmentation yet. I found iteratively adding content to the deliverable and incrementally improving different parts of the design helps as you go over a part multiple times and may catch any mistakes that may have slipped through. Reviewing the lecture slides from our previous courses also helped.

3. **Which of your design decisions stemmed from speaking to your client(s) or a proxy (e.g. peers, stakeholders, users)?**

   We made most of our design decisions keeping our stakeholders in mind. One feature we added explicitly because of stakeholder feedback was the Risk Policy feature which asks for explicit user confirmation for tasks deemed to be high risk. This was done after multiple peers brought up their concerns about the application accidentally making mistakes while converting speech to text.

4. **While creating the design doc, what parts of your other documents (e.g., requirements, hazard analysis, etc), if any, needed to be changed, and why?**

   I feel we did a fairly good job of coming up with hazards as none of the risks that were brought up in early testing or meetings were something we hadn't already considered. However, our SRS could use some refining as some requirements can be interpreted in a few different ways. We will create issues to address these requirements to make them more concise and simple to understand.

5. **What are the limitations of your solution? Put another way, given unlimited resources, what could you do to make the project better? (LO_ProbSolutions)**

   The current limitation of our solutions is using off-the-shelf existing AI models for our project. Given unlimited time, we would aim to create our own model as our use-case is relatively specific and niche. Making API calls for every single action will quickly get very expensive and will make scaling the application challenging over time. Our error-handling strategy is also very rudimentary at the moment; it simply asks the user to repeat or rephrase. Smarter and more automated recovery mechanisms could be added, given additional resources. Adding more test cases would certainly help with the robustness of the application but testing requires significant resources, particularly on different platforms as our application will be supported on multiple platforms. Adding some local processing would also be beneficial as the application in it's currently design will not work without an active internet connection.

6. **Give a brief overview of other design solutions you considered. What are the benefits and tradeoffs of those other designs compared with the chosen design? From all the potential options, why did you select the documented design? (LO_Explores)**

   One of the alternatives we considered was building a rule-based system where voice commands would be mapped to structured templates for specific tasks and if a task could not fit one of the templates, it would not be attempted. This makes the system very predictable and stable but maintaining it and extending it is very cumbersome.

   Another alternative we considered was a fully machine learning driven pipeline where intent detection, safety handling and action were all handled by a single model. This allowed the system to be very adaptable for a wide variety of different uses but makes it very difficult to add safeguards. Testing for this method is also quite difficult as it is

only feasible to test a certain number of cases; and quite a few different types of cases could go untested.

We considered a plugin system as well where the user could download and install plugins for certain applications. This would make extending the system over time easy but added a lot of complexity to dependency management, version management and cross plugin communication. Not to mention it would require more steps from users to setup as well. We ended up dropping the plugin idea completely due to these reasons.

In the end we ended up picking a balance between structure and flexibility. It keeps components sepearate enough for extensibility while allowing us to maintain control over the application and testing.

Gourob Podder Reflection

1. **What went well while writing this deliverable?**

   This deliverable was a lot less hectic in terms of planning. We did a large chunk of the MIS before the TA meeting which allowed us to get good feedback on the deliverable before submission and clear up misconceptions.

2. **What pain points did you experience during this deliverable, and how did you resolve them?**

   The main paint point during this deliverable was making sure we weren't being too specific in how we wrote the modules like including specific services, APIs or code implementations. I primarily used the good examples defined by the prof to understand the expectations of this section and read the relevant slides.

3. **Which of your design decisions stemmed from speaking to your client(s) or a proxy (e.g. peers, stakeholders, users)?**

   The security gate module was a direct result of feedback from our peers and stakeholders. Many users expressed concerns about accidental destructive actions being performed by the application due to misinterpretation of voice commands. To address this, we incorporated a safety module that classifies actions based on their risk levels and prompts users for confirmation when necessary.

4. **While creating the design doc, what parts of your other documents (e.g., requirements, hazard analysis, etc), if any, needed to be changed, and why?**

   For the most part our docs were fairly consistent. However we did notice some requirements which could reworked to be more specific and measurable to be effectively mapped to modules.

5. **What are the limitations of your solution? Put another way, given unlimited resources, what could you do to make the project better? (LO_ProbSolutions)**

In an ideal scenario we would create our own model as there are a specific class of llms called large action models (LAMs) which are specifically designed for action planning tasks. These models are optimized to understand and execute complex sequences of actions based on user commands. These models are typically smaller and more efficient than general-purpose LLMs, making them suitable for low latency and better user experience.

6. **Give a brief overview of other design solutions you considered. What are the benefits and tradeoffs of those other designs compared with the chosen design? From all the potential options, why did you select the documented design? (LO_Explores)**

   One of the alternatives we considered was a complete local system. This would have ensured better privacy and security for users as all data would be processed locally. However, this would have required significant computational resources and additional technical expertise in being able to train SOTA models. In addition, the given timeline for the capstone we would have severely limited the features we could implement and hurt the overall user experience.

   We also considered a monolothic model design where we would use a very large multimodal LLM to handle all aspects of the application from speech to text, intent detection, action planning and execution. While this would have simplified the architecture of the application, it would have made it very difficult to add safeguards and test the system effectively. The api costs for such models are also very high and would have made scaling the application challenging.

   In the end we chose a modular design which balanced focus for usability and accessibility while also being able to get great results by using a mixture of models approach through our modular architecture.

Team Module Reflection

1. **After you have implemented another team's module, which means this isn't filled in until after the original deadline). What did you learn by implementing another team's module? Were all the details you needed in the documentation, or did you need to make assumptions, or ask the other team questions? If your team also had another team implement one of your modules, what was this experience like? Are there things in your documentation you could have changed to make the process go more smoothly for when an "outsider" completes some of the implementation?**

   By implementing another team's module, we learned how dependent a correct implementation is on clear contracts and integration details, not just the routine list. When the implementer is not the author, any ambiguity turns into extra time spent reading, guessing intent, or coordinating with the other team. This also made it clear that

"small" missing details (types, validation rules, and ownership of state) can change the architecture of the implementation.

For the module we implemented (Text Buffer), the other team's MIS/MG were helpful for understanding the intended responsibilities and the core API. In particular, the documentation clearly stated the main operations (create, modify, move, delete), the state concept (a mapping from TextID to TextObject), and the exceptions expected (Invalid Location, Text Not Found). However, not all the details needed for a clean, drop-in implementation were present. We still had to make assumptions in order to complete it. Examples include:

- What exactly counts as a valid location (finite coordinates only, canvas bounds, snapping, negative values, etc.).
- The concrete structure of `location` and `formatting` in code, and whether formatting is stored inside TextObject or handled by another part of the system.
- ID strategy details: whether TextID must align with other object IDs, whether IDs must persist through undo/redo, and how uniqueness is guaranteed across sessions.
- How this module is expected to interact with global state, because the docs referenced usage of geometry state, but it was unclear whether TextBuffer directly mutates state or reports changes for a mutator layer to apply.

Because these details were not fully specified, we either chose the most reasonable interpretation (based on the MIS and the surrounding system design), or we would need to ask the other team to confirm expectations before final integration. In short, the documentation provided the "what" but not always the "how" needed for consistent integration.

Our team also experienced the reverse situation when another team implemented one of our modules, the Proxi UI interface. Their feedback made the documentation gaps very obvious. They reported that there were no API endpoint implementations and no clear guidance on UI and backend integration. They also noted that there was no specific documentation for UI flows and design guidance. Finally, they pointed out an important scope issue: UI is not just a small module, it is a full feature that is often critical and requires continuous integration and development, which can be beyond the scope of implementing a single module from MIS/MG alone. Even with these challenges, they still made a best-effort implementation to align with our MIS, MG, and SRS, and they asked us to add clarifications or requirements through the PR thread.

This experience was valuable because it showed us what an outsider needs most: explicit integration contracts, concrete data formats, and examples. If we could revise our documentation to make outsider implementation go more smoothly, we would make the following changes:

- Add a clear UI-to-backend integration section, including endpoint list, request/response schemas, error cases, and authentication or state assumptions.
- Document UI flows explicitly: key screens, user actions, expected state changes, and any navigation or validation rules.
- Provide a small set of acceptance tests or scenarios ("given/when/ then" style) so outsiders can verify behaviour without guessing.
- Clarify module boundaries and ownership of state (what the UI must implement versus what depends on other subsystems), and explicitly list any dependencies that must exist first.
- Include minimal reference implementations or stubs (for example, mock endpoints or placeholder services) so the UI can be developed and tested before the backend is complete.

Overall, implementing another team's module and having another team implement ours reinforced the same lesson: MIS/MG documentation must be precise about assumptions, types, validation, and integration. Without that, outsiders can still implement something, but they will need to make assumptions, ask questions, and iterate more through PR feedback.