1 — High-level architecture & recommendation

Goal: a modular dialogue system that supports:

Characters (metadata + relations),

Scenes (level/scene containers for dialogues),

Dialogues (nodes, lines, speaker),

Choices (visible options),

Choice impacts (flags, numeric changes, branching, endings),

VO & localization friendly,

Editor-friendly authoring workflow.

Two realistic approaches (pick one):

A — ScriptableObjects + custom node graph editor (Unity)

Pros: full control, tightly integrated with Unity, easy to serialise, great for small teams/academic projects.

Cons: building a good node editor is extra work; authoring is manual without a text-based DSL.

B — Use a narrative DSL runtime + Unity integration (recommended) — Yarn Spinner or Ink (both integrate well with Unity).

Pros: author-friendly (writers can write stories in text files), proven for branching narratives, good community & tooling. Yarn has Unity packages and importers; Ink is used in many commercial games. Integrates with Unity for variables, commands, and triggers. For faster development, the commercial Pixel Crushers “Dialogue System for Unity” is a powerful, albeit paid, solution.

My recommendation for this FYP: Use Yarn Spinner or Ink as the primary dialogue authoring format and couple it with a small set of Unity ScriptableObjects for Character metadata and Scene bindings. This gives the best balance: writing-friendly plus Unity runtime control. See Yarn/Ink docs for variables and choices.

Why Yarn/Ink? They allow branching, variables, conditions, and are easier for writers to iterate. If time is extremely limited, Pixel Crushers' asset accelerates everything but costs money and is heavier.

Also follow these best practices for branching complexity (don't over-branch; use flags & illusion of choice where needed).

---

2 — Data models (authoring + runtime)

All models below are shaped for Unity. Use ScriptableObject for authoring character metadata and scene bindings. Use Yarn/Ink files (or JSON) for the actual dialogue content.

2.1 CharacterData (ScriptableObject)

Holds static character info and runtime links.

Fields:

string id (unique key)

string displayName

int age

enum Gender {Male,Female,Other,Unknown}

string shortDescription

string background (longer text)

Sprite portrait (or reference to 3D model/bust)

AudioClip[] voiceClips (optional)

List<CharacterRelation> relations (see below)

bool isPlayer (if player-controlled actor)

Color uiColor (UI tint)

```
[CreateAssetMenu(menuName="WhisperingGate/CharacterData")]
public class CharacterData : ScriptableObject {
    public string id;
    public string displayName;
    public int age;
    public Gender gender;
    [TextArea] public string shortDescription;
    [TextArea] public string background;
    public Sprite portrait;
    public AudioClip[] voiceClips;
    public CharacterRelation[] relations;
    public bool isPlayer;
    public Color uiColor = Color.white;
}

[System.Serializable]
public struct CharacterRelation {
    public string targetCharacterId; // id of other character
    public RelationType relationType; // enum: Sibling, Friend, Enemy, Mentor, etc.
    [Range(-100,100)] public int weight; // -100 hostile to +100 friendly
}
```

2.2 Dialogue data (authoring)

If using Yarn/Ink: dialogues live in .yarn / .ink files, referencing actor: characterId and choices. Yarn/Ink support variables/conditions.

If using ScriptableObjects: structure like DialogueTreeSO containing array of DialogueNodeSO.

Minimal Dialogue Node model (ScriptableObject-style)

string nodeId

string actorId (speaker)

string lineText

AudioClip voice (optional)

Choice[] choices

bool isEndNode

List<string> commands (runtime commands to run when this node is activated: set flag, trigger timeline, add item)

```
[System.Serializable]
public class Choice {
   public string choiceText;
   public string nextNodeId; // id of the node to jump to
   public ChoiceImpact[] impacts;
}

[System.Serializable]
public class ChoiceImpact {
   public string variableName; // "trust_alina"
   public ImpactType type; // Set, Increment, Decrement, Toggle
   public int intValue;
   public string stringValue;
}
```

But if you use Yarn/Ink, you can encode conditions & variable changes inline:

Yarn example:

```
<<set $trust_alina += 10>>
Alina: "Thank you."
-> continue
```

(Use Yarn/Ink for brevity and writer friendliness.)

2.3 Scene model

A SceneBinding ScriptableObject maps a Unity Scene to a set of dialogue nodes / Yarn file entry points.

string sceneId

SceneAsset scene

TextAsset yarnFile (or DialogueTreeSO)

string startNode (entry point)

List<DialogueTrigger> triggers (positions in level mapped to node IDs)

2.4 Dialogue Triggers (level objects)

Placed in the 3D world (prefab) — when the player enters or interacts, it calls DialogueManager to start a node. Fields:

string triggerId

string startNodeId

TriggerType {OnEnter, OnInteract, OnLookAt}

bool singleUse

bool interruptPlayerMovement

---

3 — Runtime Systems & Flow

Overview diagram (conceptual):

```
Player -> Interaction -> DialogueManager -> DialogueRuntime -> UI / Audio / Commands ->
GameState / FlagManager
                                        ↑
                                 ScriptableObjects/Yarn Files
```

3.1 DialogueManager (singleton)

Responsibilities:

Load dialogue content (Yarn/Ink or ScriptableObjects).

Maintain current node, queue VO, run node commands.

Evaluate choice conditions (if a choice requires $trust_alina > 50).

Fire events: OnDialogueStarted, OnNodeChanged, OnChoiceSelected, OnDialogueEnded.

Save dialogue runtime state to SaveManager (for multiple sessions).

Key features:

Support skippable lines (for testing/VO).

Support interruptions (cutscenes or higher-priority events).

Provide hooks for the UI to retrieve the current line and available choices.

3.2 UI Layer

Dialogue panel (text box), speaker name, portrait, choice buttons.

Typewriter effect optional, with speed control.

Choice button pooling for dynamic number of choices.

Support for linking choices to gameplay (e.g., disable movement).

3.3 Flag / GameState Manager

Stores variables that choices change (booleans, ints, strings).

Must be serializable for Save/Load.

Provide API: SetFlag(string key, bool), AddInt(string key,int), GetInt/Bool.

Support per-character relationships (trust, affinity) as numeric values. For example: GameState.AddInt("trust_alina", 10).

3.4 Command/Effects system

When a node executes, it can include commands that call other runtime systems:

TriggerTimeline("writer_house_cutscene")

GiveItem("journal")

ModifyRelationship("alina", +10)

SetEnding("GoodEnding")

Commands can be implemented as strings parsed by a CommandDispatcher, or via events.

3.5 Save/Load

Save GameState (flags & variables), current scene, player position, and current branching progress.

Save at checkpoints (after key choices) and allow continuation.

For academic demo, a single quicksave slot is enough.

3.6 VO & Lip Sync

VO Clips tied to nodes (or use TTS in early prototype).

DialogueManager will play AudioClip and lock next node until VO ends (or allow skip).

Optionally use simple viseme blending or face animation for 3D characters (can be deferred if time-limited).

---

4 — Authoring & Tools

Option A — Yarn Spinner / Ink

Write dialogue in Yarn or Ink files; they support variables, conditional branching, and commands. Yarn integrates with Unity and has importers to map actors to Unity characters. Use Yarn for classic dialogue with choices; Ink for more programmatic stories.

Workflow:

Writer edits .yarn file.

Designer / developer assigns actor names that map to CharacterData id in Unity.

DialogueManager parses Yarn and pushes nodes to UI.

Use Yarn commands to modify variables (e.g., <<set $trust_alina += 10>>) which map to GameState.

Option B — ScriptableObjects + Custom Node Editor

Create DialogueNodeSO and DialogueTreeSO ScriptableObjects.

Build small editor windows or use UnityEditor.Experimental.GraphView for a node-based editor (more dev time).

Good control, but requires more tooling work.

Option C — Pixel Crushers Dialogue System (Asset Store)

Powerful and feature-rich; supports localization, timelines, complex behaviors — good if budget and time allow.

---

5 — Example Unity code (skeletons)

Below are compact skeletons you can paste into Unity and expand. They assume C# in Unity 2021+.

5.1 CharacterData ScriptableObject

```
using UnityEngine;

[CreateAssetMenu(menuName = "WhisperingGate/CharacterData")]
public class CharacterData : ScriptableObject {
   public string id;
   public string displayName;
   public int age;
   public Gender gender;
   [TextArea] public string shortDescription;
   [TextArea] public string background;
   public Sprite portrait;
   public AudioClip[] voiceClips;
   public CharacterRelation[] relations;
   public bool isPlayer;
```

```
    public Color uiColor = Color.white;
}

public enum Gender { Unknown, Male, Female, Other }

[System.Serializable]
public struct CharacterRelation {
    public string targetCharacterId;
    public RelationType relationType;
    public int weight;
}

public enum RelationType { Sibling, Friend, Enemy, Mentor, Stranger }
```

5.2 Minimal DialogueManager (pseudo)

```
using System;
using System.Collections.Generic;
using UnityEngine;

public class DialogueManager : MonoBehaviour {
    public static DialogueManager Instance { get; private set; }

    public event Action OnDialogueStarted;
    public event Action OnDialogueEnded;
    public event Action<DialogueNode> OnNodeChanged;

    private IDialogueProvider provider; // Yarn/Ink or SO provider
    private DialogueNode currentNode;

    void Awake() {
        if (Instance != null) Destroy(gameObject);
        Instance = this;
    }

    public void StartDialogue(string dialogueId, string startNodeId = null) {
        provider = DialogueProviderFactory.GetProvider(dialogueId);
        currentNode = provider.GetNode(startNodeId ?? provider.GetStartNode());
        OnDialogueStarted?.Invoke();
        ShowNode(currentNode);
    }

    void ShowNode(DialogueNode node) {
        currentNode = node;
        // send node to UI
        OnNodeChanged?.Invoke(node);
        // handle auto-commands
        foreach (var cmd in node.commands) CommandDispatcher.Dispatch(cmd);
    }

    public void SelectChoice(int index) {
        var choice = currentNode.choices[index];
        // apply impacts
        foreach (var imp in choice.impacts) GameState.ApplyImpact(imp);
        // jump to next node
        var next = provider.GetNode(choice.nextNodeId);
        if (next == null) { EndDialogue(); return; }
        ShowNode(next);
    }

    void EndDialogue() {
        OnDialogueEnded?.Invoke();
    }
}
```

DialogueNode, DialogueProviderFactory, CommandDispatcher, GameState are components you implement per spec above.

---

6 — Example data snippets

Yarn example (small)

```
title: Jungle_Start
---

=== start ===
-> node_intro

=== node_intro ===
<<actor "protagonist">>
I opened my eyes. The sky is wrong.

<<actor "narrator">>
You can go after the scream or inspect the journal.
```

```
+ "Go after the scream" -> screamPath
+ "Inspect the journal" -> journalPath

=== screamPath ===
<<set $courage += 1>>
You run toward the scream...
-> END

=== journalPath ===
<<set $knowledge += 1>>
You pick up the burnt journal. Its pages hum.
-> END
```

This shows how variables are changed inline; Unity will sync Yarn variables with GameState so GameState.GetInt("knowledge") reflects Yarn changes.

---

7 — Choice impacts, branching & endings

Principles to avoid combinatorial explosion:

1. Use small, persistent variables (trust, guilt, courage) that many choices modify. Endings are computed from these variables (thresholds), not by enumerating every path. This is how many narrative games (Telltale, Life is Strange) avoid exponential state blow-up.

2. Make some choices cosmetic / local — only a few choices are "major" and affect endings; the rest add flavor and incremental variable changes.

3. Use flags for binary events (e.g., saved_writer, burnt_journal_found) and use those flags to gate scenes and dialogue.

4. Compute ending by checking key variables at the final ritual (e.g., if trust_alina>50 and courage>20 then GoodEnding else BadEnding). Avoid making ending dependent on dozens of flags.

---

8 — Integration: Timeline, cutscenes, triggers

Use Unity Timeline to create short cinematic sequences; trigger Timeline via CommandDispatcher.TriggerTimeline("writer_house_entry").

DialogueManager should be able to pause player control (set a PlayerController.enabled = false) during cutscenes or timed interactions.

Use DialogueTrigger prefabs to place nodes in the world; those triggers can be OnEnter or OnInteract (player presses E).

---

9 — Localization, VO, and QA

Localization: Keep dialogue text outside of code (Yarn/Ink/JSON). For ScriptableObjects, add localization keys instead of raw text. This enables later translation and also lets you swap text for testing.

VO: Keep VO clips per node using a naming convention. For production, store timing metadata (start time/length) to sync subtitles.

QA: Provide a debug console to set variables at runtime (e.g., GameState.Set("trust_alina", 100)) so testers can jump to endings quickly.

---

10 — Performance & Scalability

Dialogue text and small JSON/Yarn files are tiny — not a performance risk. The main cost is building many nodes and keeping huge graph editors in memory. For large projects, stream only current scene's dialogues.

When using ScriptableObjects, avoid loading all SOs into memory; group them by SceneBinding and load on demand.

---

11 — Testing checklist

Verify variable syncing between Yarn/Ink and GameState.

Test choice branches that change the same variable in different ways (race conditions).

Validate save/load: choose options → save → quit → load → confirm state preserved.

VO sync: start dialogue with VO → skip → ensure VO stops.

Edge cases: player moves while a dialogue is running; overlapping triggers.

---

12 — Example implementation roadmap (6 weeks for a 15-minute demo)

Week 1 — Foundation

Create CharacterData SOs for main cast (Protagonist, Alina, Writer, Mysterious Woman).

Implement PlayerController (first person recommended for horror).

Implement DialogueManager skeleton and UI mock.

Week 2 — Dialogue authoring & triggers

Integrate Yarn/Ink, author a 1–2 minute dialogue (intro).

Implement DialogueTrigger prefab and SceneBinding.

Week 3 — GameState & Choice impacts

Implement GameState (flags & numeric vars).

Implement ChoiceImpact system and command dispatcher.

Author a branching node that modifies variables.

Week 4 — Scenes & environment

Build jungle + writer’s house small level. Add triggers for 3–5 interactions.

Implement camera cutscenes (Timeline) for 2 transitions.

Week 5 — VO & Sanity effects

Add ambient audio & whisper zones.

Add screen post-processing tied to sanity variable.

Week 6 — Polish & testing

Fix bugs, implement save/load, create build, and record playtest notes.

---

13 — Example: mapping to your provided GDD & script

Use each Scene in the GDD as a SceneBinding entry. For example:

SceneBinding: JungleWakesYou → Yarn entry node_intro.

Add DialogueTriggers at the twisted tree, near the dolls, and at the journal.

Map characters to CharacterData with relations (Protagonist ↔ Alina: Sibling, weight +80).

Implement three major variables to compute endings: trust_alina, ritualFoundness, courage. Use thresholds to compute the final ending at the ritual scene.

Keep the first playable chapter (the uploaded FYP-SS.docx scenes 1→11) as your 15-minute demo: implement the exploration & critical choices from those scenes.

---

14 — Quick comparison & references (why I recommended Yarn/Ink and the practices above)

Yarn Spinner & Ink are used widely for branching narrative because they separate authoring from code and handle variables / conditions elegantly. Yarn has Unity integration for mapping actors and running commands; easier for writers.

Pixel Crushers' Dialogue System is a very powerful turnkey asset if you want features fast (localization, timeline, node editor), but it's paid.

Best practices: avoid exponential branching; use small persistent variables and a limited set of "major decisions" to determine endings; this mirrors how Telltale and Life Is Strange handle choices (lots of small choices, few major branching forks).

---

15 — Deliverables I can give next (pick one)

Full Unity-ready ScriptableObject templates + C# code files (ready to paste into project) covering CharacterData, DialogueNode SO, DialogueManager, GameState, and DialogueTrigger.

A sample Yarn file that implements the first chapter from FYP-SS.docx, with variables and impacts wired.

A Trello-style task list per team member and week (tested roadmap).

A short tutorial (step-by-step) to hook Yarn into Unity and sync Yarn variables to GameState.

Which of the above do you want me to produce now? (I can generate the ScriptableObject C# files + a sample Yarn scene and a sitemap mapping the 1st chapter to nodes — ready to drop into Unity.)