Offline AI NPC — documentation

Every page of `Documentation`, screenshots inlined. One file, no network.

Pages

Offline AI NPC — documentation

Fully offline conversational NPCs for Unity: local language model, local speech recognition, local speech synthesis, per-character memory and state. No cloud, no API keys, no per-token cost, no telemetry.

Start here

Is this for your game?	**Read first.** What it fits, what it does not, with measured numbers
Getting started	Import to a talking character
Beginner's guide	The long version: empty project to a build you can hand to somebody, assuming nothing
Recipes	Five shapes this is good at, and roughly what each costs to build

Building with it

Dialogue	Wiring a talking character — the complete controller
State	Moods, needs, relationships. **Opens with the ten-line version**
Perception	The character knowing what is actually around her
Actions	The character doing, not only saying
Languages	How many languages, honestly — and how to add one

Before you ship

Limitations	Every limit, with its mitigation where one exists
Hardware tiers	What each tier means, and what happens below the lowest
Licensing	What you may ship and what shipping it obliges
Troubleshooting	Real error strings mapped to fixes

Reference

Architecture	Assembly boundary, seams, folder layout
Third-party notices	Full inventory
Changelog	

The four numbers worth knowing before you read anything else

Measured on an RTX 3060 Laptop (6 GB), Qwen3-4B, all 36 layers on the GPU. The tooling that produced them ships with the package, so you can produce your own.

Time to the player hearing the first word	**2.0 s** median (p10–p90 1.5–2.8 s)
First reply of a session	**~2.1 s** — the prompt cache filling
Contention: the same reply, with background characters generating	2.4 s → **12.7 s**
State descriptors colouring the model's speech	**~6 times in 10** — the deterministic half is exact

Everything in this documentation traces to a measurement or says that it does not.

What is not included

Named so you do not go looking: RAG or document retrieval, lip sync, emotion control in speech synthesis, streaming (as opposed to per-utterance) speech recognition, and mobile, web or console support.

Offline AI NPC

A fully offline conversational-NPC stack for Unity: local LLM dialogue, local speech-to-text, local text-to-speech, and per-character memory that survives scene loads and restarts. No cloud, no API keys, no per-token cost.

This folder ends in `~`, so Unity excludes it from the asset database and from compilation — documentation lives here and never becomes an import cost.

Layout

```
Runtime/
  Core/        Language, ILanguageSource, external-process lifecycle, dependency guards
  Speech/Tts/  ITTSEngine + Piper and VOICEVOX engines + the switching manager
  Speech/Stt/  the always-hot mic loop and the per-NPC utterance recorder
  Memory/      rolling per-NPC conversation memory over an injectable key/value store
  Hardware/    hardware tier detection (what this machine can afford to run)
Editor/        editor-only lifecycle hooks
Tests/         EditMode and PlayMode suites — no model, mic, or network required
LICENSES/      third-party notices
```

The boundary

`OfflineAINPC.Runtime` has an empty `references` list and an empty `precompiledReferences` list. Beyond the engine it takes **nothing at all** — not the host game's assembly, not LLMUnity, not whisper.unity, and no third-party library. It used to take Newtonsoft.Json, and that one line cost every buyer without it 36 compile errors on import; the JSON it needs is now `Core/Serialization`, about four hundred lines that belong to the package. If it compiles, it is portable — that is the entire point of the assembly definition.

Everything the plugin cannot know is injected by the host:

Seam	Interface	The game supplies
Persistence	`IKeyValueStore`	a PlayerPrefs-backed store
Current language	`ILanguageSource`	a wrapper over its own language manager
Piper voice casting	`IPiperVoiceRouter`	optional; serialized fields are the default
Loading UI	`TTSEngineManager.OnSwitching` / `OnSwitched`	a listener that drives its overlay

External processes

Piper (and VOICEVOX) run as OS child processes. A Unity domain reload nulls the static fields holding them but does **not** kill the processes, so every host that spawns one registers with `ExternalProcessRegistry`. Assembly reload, play-mode exit and application quit all drain it. Enter Play Mode with "Reload Domain" disabled is handled explicitly: the registry is cleared on `SubsystemRegistration` rather than assumed empty.

Dependencies

LLMUnity and whisper.unity are installed by the user, not bundled — bundling either is a duplicate-assembly compile error for anyone who already has them. Integration points are guarded by `LLMUNITY_PRESENT` / `WHISPER_PRESENT`, declared as `versionDefines` in `OfflineAINPC.Runtime.asmdef`.

Running the tests

Window → General → Test Runner. Both suites run with no model present: the LLM, TTS and STT sit behind interfaces and the tests supply their own fakes.

Is this for your game?

Read this before you buy. It is written to talk the wrong buyer out of the purchase, because a refund and a one-star review cost more than one sale is worth — and because the shape of this plugin genuinely suits some games and genuinely ruins others.

Every number below was measured on real hardware and is reproducible with the tools in the package. Where something is unmeasured, it says so.

The one thing to understand first

All characters share one local model server, and requests through it are serialised.

Everything on this page follows from that sentence. A local language model is a single process holding several gigabytes of weights; you cannot run four of them, and requests to the one you have queue behind each other.

Measured on an RTX 3060 Laptop (6 GB) with Qwen3-4B, median time to the player hearing the first word:

Situation	Median
One character, shipping path, nothing else generating	**2.0 s**
One character, nothing else generating, waiting for the *whole* reply	2.4 s
Waiting for the whole reply **while other characters generate in the background**	**12.7 s**

The two lower rows are the honest comparison — same measurement mode, so the difference is contention and nothing else: **background conversation costs roughly 5×**. The 2.0 s figure is the path a shipping game uses, where speech begins at the first finished sentence instead of at the end of the reply.

No amount of tuning removes the 5×; it is the queue. The mitigation is in Limitations: throttle background conversation while the player is speaking. The pattern is a few lines and it works, but you have to write it.

Fits well

A small cast. One to a handful of characters. A companion, a shopkeeper, a suspect, a roommate. One conversation at a time is the design centre.

Player-initiated, turn-shaped conversation. The player speaks or types, the character answers. This is a dialogue system, not a barking system.

Genres built around talking. Visual novels. Life and social sims. Interrogation and detective scenes. Companion characters. VR social experiences. Immersive sims with a few deep NPCs rather than many shallow ones.

Games that already gate on hardware. VR titles, mid-to-high spec desktop. Your audience has already accepted a GPU requirement, so the model's is not a new barrier.

Offline and privacy as requirements. No API keys, no per-call billing, no rate limits, no network at all. Nothing leaves the machine, and there is no telemetry in this package — not even anonymous. If your game must work on a plane, in a school, or under a privacy policy that forbids sending player speech to a third party, that is the case this was built for.

Does not fit

Plainly, so nobody is surprised after paying.

Mobile, web, or console. Desktop only: Windows, macOS, Linux. This is not a roadmap item — it is a consequence of shipping a multi-gigabyte model and external speech processes.

Crowds of talking NPCs. Several simultaneous conversations push a player's own reply from 2.0 s to a measured 12.7 s. A tavern of twenty characters chatting in the background is not something this can do, and throttling them means they are not really talking.

Real-time combat barks. The latency budget for "enemy spots you and shouts" is a few hundred milliseconds. The median here is 2.0 s to first audio and the first line of a session takes about 2.1 s. Use audio clips for barks; they are better at it.

Tightly authored narrative. The grammar constrains the *shape* of a reply — that it is one sentence, that an action payload is well-formed — never its content. If a line must be delivered word for word because the plot turns on it, a dialogue tree is the right tool. **A dialogue system and this plugin are complements, not competitors:** author the beats that matter, and use this for the space between them.

Low-spec target audiences. See Hardware tiers. Below the lowest tier the plugin degrades to no language model at all, which is a working state but not the product you are buying.

Anything needing guaranteed output. A language model produces a different reply every time, occasionally a bad one. There is a resilience layer so the game never crashes, but if your design cannot tolerate an off-key line, this is the wrong tool.

Fits with care

Many characters where only one is active at a time. A dormitory of six where you talk to one is fine. Six talking at once is not. The distinction is whether they generate simultaneously, not how many exist.

Games leaning on the deterministic half. State axes, thresholds and events are exact every time and cost nothing; the language model is the expensive, probabilistic part. A design that drives most of its behaviour from

state and reaches for the model sparingly gets the best of both. See State.

A first project. It works, but you will be integrating an LLM, a speech recogniser and a speech synthesiser, each with its own failure modes. Budget time for that rather than expecting a drop-in NPC.

What you also need

Two external dependencies, neither bundled, both free:

- **LLM for Unity** (LLMUnity), for the language model.
- **whisper.unity**, for speech recognition. Optional if you only want typed input.

And a model file — 1 to 5 GB depending on tier, which you or your player downloads. The package ships a catalogue of licence-checked options and a download manager, but not the weights themselves.

Speech synthesis is an external process you install (Piper, or VOICEVOX for Japanese). If you redistribute Piper with your game, note that it links espeak-ng under GPL-3.0 — see Licensing, which explains exactly what that obliges and how to avoid it.

Still unsure?

Install a sample, press Play, and read your own numbers: `Tools ▸ Offline AI NPC ▸ Live verification` times a real turn on *your* machine — first audio, tier, layers on the GPU. "Will this run acceptably here" is the one question no cloud-based competitor can answer for you, and the only honest way to answer it is on your hardware.

Beginner's guide

The long way round, assuming you have never installed a Unity package with external dependencies. It starts with an empty project and ends with a build you can hand to somebody else. Allow an hour; most of it is downloads.

A Russian translation is kept in the project repository under `docs/ru/BeginnerGuide.md`.

---

0. What this is, in one paragraph

Your character talks. The language model, the speech recognition and the speech synthesis all run **on the player's own machine** — no cloud, no API keys, no per-token bill, no telemetry, and it works with the network cable pulled out. What you install is: a model runner, a model file, a voice, and this plugin to hold them together.

What you will have at the end of this page: a scene where you type or speak a question and a character answers you out loud, and a build of it that runs on a machine without Unity.

---

1. Before you start

Unity	6000.5.7f1 is what this is developed and tested on. 2022.3 LTS or newer should work, but only 6000.5 is verified
Render pipeline	any — Built-in, URP, HDRP. The runtime references no rendering package
Platform	Windows, macOS or Linux **desktop**. Not mobile, not web, not console, and that is a hard limit — the model is gigabytes and the speech engines are separate processes
Disk	10 GB free is comfortable
Graphics	runs with no GPU at all, just slower. 6 GB of VRAM fits a 4B model entirely on the card
Internet	for the downloads on this page only. Never at runtime

If any of those are a problem, read Is this for your game? before spending the hour.

---

2. Make a project

New project, any template. Nothing here depends on which.

Adding this to a project you already have is fine too: no step on this page edits your Project Settings, your scenes or your existing assets.

---

3. Install LLM for Unity — required

This is the thing that actually runs the language model. Without it your character cannot form a sentence.

- Package id: `ai.undream.llm`, version **3.0.3** or newer
- Free, and not bundled here — it is somebody else's package under its own licence, and freezing a copy inside this one would pin you to whatever version shipped that day
- Install it the way its own documentation says

How you know it worked: *Window ▸ Package Manager* lists **LLM for Unity**.

If Unity was already open, it may take one recompile before this plugin notices. You do not need to restart — `Tools ▸ Offline AI NPC ▸ Re-check LLMUnity installation` forces the check.

---

4. Install whisper.unity — optional

This is speech **input**: the player speaking instead of typing. Skip it and everything else works; typed input is not a lesser path.

Window ▸ Package Manager ▸ + ▸ Add package from git URL, and paste:

```
https://github.com/Macoron/whisper.unity.git?path=/Packages/com.whisper.unity
```

A missing dependency is never a compile error here. Features that need one are compiled out. If you skip this, the push-to-talk code simply is not there.

---

5. Import Offline AI NPC

Import the package.

On the first editor load afterwards a **Welcome** window opens by itself, with three cards in the order the questions come up: are the dependencies installed, is there a model, is there a scene to start from.

- It appears **once** and never again — not after a recompile, not in your next project.
- It changes nothing on its own. Everything it can do is behind a button you press.
- Want it back? `Tools ▸ Offline AI NPC ▸ Welcome`.

---

6. Download a model

`Tools ▸ Offline AI NPC ▸ Model Manager`.

Every row tells you its **tier**, its **languages** and its **licence** before you download anything. The default line-up is chosen so that nothing you ship needs an attribution.

Pick the one that matches your language and your hardware tier, and download it.

Things worth knowing:

- The manager verifies the finished file against a hash. A half-finished download is caught **here**, not three hours later as inexplicable behaviour.
- `[Ready]` means the file is on disk *and* its header parses. `[Manual]` means you supplied it yourself, so the manager can only tell you what the file itself says.
- **Download one.** They are gigabytes each, and it is very easy to end up with six.

---

7. Add a voice

Speech **output**. The plugin ships the integration; the engine is yours to install.

1. Download Piper and put its folder somewhere your project can reach.
2. Add at least one voice — **both files**: `name.onnx` **and** `name.onnx.json`.

> ### The most common setup failure in this whole package
>
> A voice with the `.onnx` but not the matching `.onnx.json`. The result is **silence, with no error message at all**. If your character never speaks and the console is clean, check this before anything else.

Japanese needs an engine you supply — VOICEVOX is the usual choice, and connecting it is about eighty lines against one interface. Worked example in Languages.

---

8. Check the setup before you build anything on it

`Tools ▸ Offline AI NPC ▸ Check my setup`.

Every line is a real check against your project, and a failure tells you what to do rather than what went wrong.

Fix everything red now. Each red item is something that would otherwise reappear later as behaviour you will not connect to its cause.

---

9. Your first conversation

Open the `Samples/HelloNpc` sample — the smallest one that talks. Press Play, type something, hear an answer.

A healthy start prints these lines. Learn to recognise them:

```
[OfflineAINPC] Hardware: NVIDIA GeForce RTX 3060 Laptop GPU (5996 MB VRAM, discrete), ...
[OfflineAINPC] Tier resolved: Standard — 3996 MB VRAM free after a 2000 MB game reserve
[OfflineAINPC] [TTSEngineManager] Active: PiperEngine (en)
[OfflineAINPC] Running qwen3-4b-q4km at tier Standard with 36 GPU layers
[OfflineAINPC] Startup verdict: tier Standard — ...; model qwen3-4b-q4km, 36 GPU layers
```

The startup verdict line is the one that matters. It means the whole chain resolved. If it never appears, your problem is upstream of dialogue and Troubleshooting is the next page to read.

The first reply of a session is slower than the rest. The prompt cache is filling. That is expected, it is measured, and it does not mean anything is wrong.

Normal, or broken?

The first hour with a local model is mostly learning which surprises are the technology and which are a mistake. These are the technology:

What you see	
The first reply takes several seconds, the rest do not	**normal** — prompt cache
The same question gets a different answer each time	**normal** — it is a model, not a dialogue tree
She will not repeat an authored line word for word	**normal** — author those lines yourself and use the model between them
She forgets the conversation after you stop playing	**normal** until you give memory a store
She forgets what you said ten minutes ago	**normal** — the window is the last ~20 turns per character
Her mouth does not move, she does not walk anywhere	**normal** — the plugin drives none of that
She says something about the room that is not there	**normal** without perception; the cure is Perception, not a sterner prompt
The editor sits still for seconds on the first Play	**normal** — the model is loading, once
One turn in ten is noticeably slower than the others	**normal** — see Limitations for the distribution

And these are not — each has a page that names the cause:

What you see	
Subtitle appears, no sound	**broken** — usually a voice missing its `.json` partner (Troubleshooting)
She answers in the wrong language	**broken** — voice switched, prompt not
Replies run on for paragraphs	**broken configuration** — the token ceiling and sentence limit, not the wording of the prompt
The startup verdict line never prints	**broken** — the chain did not resolve; nothing downstream will work
Push-to-talk sends things you never said	**broken** — the utterance gate is not in the path

10. What is in the plugin, and what each part is for

Everything below ships in the package. You will not need all of it on day one — most projects start with dialogue and add the rest when they meet the problem it solves. This section exists so that you know what is already there before you build it yourself.

The runtime, by area

Area	What it is for	The problem it solves
Core	languages, presets, sampling, the streaming reply parser, logging	A model answering in the wrong language, or a reply arriving as tokens when speech needs sentences
Speech ▸ TTS	the engine interface, Piper driver and process pool, voice routing, the streaming speech session	Turning a reply into audio while it is still being generated, instead of after
Speech ▸ STT	shared microphone loop, push-to-talk recorder, the utterance gate	Whisper inventing sentences out of silence — see below, it matters
State	axes, bands, traits, mood presentation, serialisation	A character whose mood and relationships change, and who reads differently because of it
Actions	the verb schema, parser, dispatcher, targets, one-shot repair	The character DOING things, not only saying them — and doing only what you allowed
Perception	points of interest, weighted queries, snapshots, phrasing	Her knowing what is actually around her, rather than being told to pretend
Memory	a small conversation window over storage you supply	Her remembering the last few turns without you inventing a format
Hardware	probe, tiers, the cascade, model catalog	Deciding at startup what this machine can carry, and picking a model to match
Models	gguf reading, asset states, storage paths	Knowing a model file is whole and usable before loading it
Diagnostics	the checks behind the setup window	Finding a broken setup at edit time rather than in a playtest
Verification	latency measurement, prompt snapshots, live checks	Proving it works, with numbers you produced yourself
Integrations	the LLMUnity grammar binding	Constraining the model's output to your action schema

Parts worth knowing about early

The utterance gate (`UtteranceGate`). Whisper hallucinates confidently on silence -- "Thank you for watching", "Subtitles by …", a lone full stop — because it was trained on video with end cards. A push-to-talk key brushed by accident therefore looks exactly like a player speaking. The danger is not a stray subtitle: if your game listens for a word as a command, an invented one triggers it, and a key brushed in passing sends the player somewhere they never asked to go. The gate judges the audio first and the text second, and every rejection says which one it was.

Action repair (`ActionRepair`). When the model returns a malformed action, one retry is allowed — and it asks for **the action alone**, never the reply again. By the time anyone knows the action was bad the speech has already been spoken, and regenerating it would make her say the same thing twice.

Traits versus state (`TraitPreset` versus `StateAxes`). Traits never move: they are why two characters with identical axes behave differently — the shy one starts lower and warms slowly. State moves. Mood is deliberately **not** persisted, because a feeling that outlives the session is not a feeling, it is a trait.

Perception weights (`PerceptionWeights`). Serialized data, never constants. The right balance of attention for a cluttered room is not the right balance for an open field, and that is a designer's call rather than the plugin's.

The language directive (`LanguageDirective`). The plugin's own contribution to your system prompt: answer in THIS language, and keep it about this long. Both sentences are written **in the language they are asking for** — an instruction the model cannot read is not an instruction.

Preset choice (`PresetCatalog`). Your entry outranks an official one for the same slot; otherwise the exact tier wins, then the closest lighter tier — because running something smaller beats running nothing.

The editor windows

All under `Tools ▸ Offline AI NPC` .

Window	What it answers
Welcome	"What do I do first?" Three cards, once, on first import
Model Manager	"Which model, and is it really here?" Tier, languages and licence per row, hash-verified downloads, and control over what enters a build
Presets	"Which model and which voice does this language and tier use?"
Check my setup	"Is anything wrong before I press Play?" Real checks, each failure saying what to do
Prompt debugger	"**Why did she say that?**" What the model was actually told. For a local product this is the only way to answer it
Live verification	"Does it actually speak?" Proves a character talks **without entering play mode** — static checks can say a configuration is valid, never that it works

What the plugin deliberately does not do

Knowing these saves you looking for them:

- **It saves nothing until you say so.** `NPCMemory` writes through a store you choose. `PlayerPrefsStore` ships with the plugin and takes one line; a file or your own save system is three methods. See Dialogue ▸ Memory. Nothing is written behind your back, because your game already owns its save format.
- **It does not move, animate or draw anything.** No subtitles, no mouth, no walking to the window when she says she is going to. The plugin decides *what* the character says and *what she decided to do*; every visible consequence is yours to wire — which is also why it works in any render pipeline and any art style.
- **It does not bundle the model runner.** LLM for Unity is a separate package under its own licence.
- **It does not bundle a speech engine.** Piper is yours to install and ship, and Japanese needs an engine you supply.
- **It does not touch your Project Settings, scenes or assets** unless you press a button that says it will.

11. Your own character

The plugin gives you the pieces; you write the turn. Here is the shape of it, with the four things beginners get wrong marked.

```
using System;
using System.Collections;
using UnityEngine;
using OfflineAINPC.Core;
using OfflineAINPC.Speech;

public sealed class TalkingCharacter : MonoBehaviour
{
    [SerializeField] private AudioSource _voice;
    [SerializeField] private int _speakerId;      // which voice, when the engine has several

    private StreamingReplyParser _parser;
    private StreamingSpeechSession _speech;
    private bool _busy;

    public IEnumerator Say(string playerLine)
    {
        // (1) ONE lock, released on EVERY exit path.
        if (_busy) yield break;
        _busy = true;

        try
        {
            _parser = new StreamingReplyParser();
            _speech = new StreamingSpeechSession(
                // (2) a Func, not a stored reference: the engine is replaced when the
                //     language changes, and during that switch it is null.
                engine:    () => TTSEngineManager.Instance != null ? TTSEngineManager.Instance.Active :
null,
                audio:     _voice,
                speakerId: _speakerId,
                onSubtitle: (text, seconds) => ShowSubtitle(text, seconds),
                onWarning:  message => Debug.LogWarning(message));

            StartCoroutine(_speech.RunSynthesis());
            StartCoroutine(_speech.RunPlayback());

            yield return GenerateReply(playerLine, onToken: textSoFar =>
            {
                // (3) CUMULATIVE or DELTA — get this backwards and she says
                //     "MMaraMara VMara VeyMara Vey, 34." instead of "Mara Vey, 34."
                _parser.AppendCumulative(textSoFar);   // LLMUnity streams cumulative text
                // _parser.Append(delta);              // a backend streaming true deltas
                while (_parser.TryDequeueUnit(out var unit)) _speech.Enqueue(unit);
```

```
            });

            _parser.Complete();
            while (_parser.TryDequeueUnit(out var unit)) _speech.Enqueue(unit);

            // (4) without this the synthesis loop waits forever for a unit that is never
            //     coming, and she never finishes speaking.
            _speech.InputComplete();

            while (!_speech.Finished) yield return null;
        }
        finally
        {
            _busy = false;      // (1) again: this is why it is a finally
        }
    }

    private void ShowSubtitle(string text, float seconds) { /* your UI */ }

    private IEnumerator GenerateReply(string playerLine, Action<string> onToken)
    {
        // LLMUnity, your own binding, or a stub. The plugin does not care which.
        yield break;
    }
}
```

The four, spelled out

1. **`_busy` released in a `finally`.** Every early return, every exception, every cancelled coroutine must clear it. The symptom when it leaks is brutal: the character stops answering **forever**, and the console says nothing.
2. **A null engine is normal.** It is a `Func<>` because the engine is swapped when the language changes. The session copes by delivering the line to the subtitle only.
3. **Cumulative versus delta.** LLMUnity's stream callback hands you the *whole reply so far* every time. Feed that to `AppendCumulative`. A backend that streams true deltas needs `Append`. The failure is silent and unmistakable once you have heard it.
4. **`InputComplete()`.** Tells synthesis no more text is coming.

Full walkthrough in Dialogue. Memory, moods and relationships are in State, which opens with a ten-line version.

---

12. The names you will actually type

The section above says what each area is for. This one says what to reach for, because a concept you cannot name is a concept you cannot look up. Only the types you touch are listed; the rest are internals that happen to be public.

Put on a GameObject

Type	What it does
`NpcStateComponent`	holds one character's axes, traits and mood
`AIPointOfInterest`	marks a thing in the world as noticeable, with an id and a description
`PerceptionQuery`	the character's own view: what she can see, ranked
`ActionDispatcher`	receives the actions she decides on and routes them to handlers
`PersistentMicRecorder`	push-to-talk recording for one listener
`TTSEngineManager`	owns the active speech engine and switches it with the language

Implement in your own code

Type	Why
`IActionHandler`	run a verb the character issued
`IActionTarget` , `IAimTarget`	be something an action can be aimed at
`IStateStore` , `IKeyValueStore`	persist state and memory in YOUR save system
`ITTSEngine` , `ILanguageAwareEngine`	add a speech engine the package does not ship
`ILlmHost`	put a different model runner underneath
`ILiveVerifiable`	let the live-verification window exercise your character

Call

Type	Why
`NpcActions.Register`	declare a verb before anything can issue it
`ActionSchema`	the grammar and the prompt block, generated from the verbs
`NPCMemory`	append turns, read back the last N
`StreamingReplyParser`	tokens in, speakable units out
`StreamingSpeechSession`	units in, audio and subtitles out
`UtteranceGate`	judge a recording, and then its transcript, before spending a turn
`UtteranceVerdict`	what the gate decided: accepted, too quiet, or likely hallucinated
`LanguageRuntime`	the current language, and the directive it contributes
`StateSerializer`	the persisted half of a character
`SetupChecks`	run the same checks the window runs, from your own code
`LatencyMeasurement`	produce your own numbers rather than trusting ours
`ActionParser`	turn the model's text into an action, or say precisely why it could not
`PerceptionSnapshot`	the ranked result of one query, reused for the prompt and the shortlist
`PoiRegistry`	every noticeable thing currently registered
`SharedMicLoop`	one microphone, several listeners, no fighting over the device
`HardwareProbe`, `TierCascade`	what this machine is, and what that means for the model
`ModelCatalog`	the recommended line-up, per tier
`AiNpcDiagnostics`, `DiagnosticsReport`	the checks and their result, as data
`LLMUnityGrammarBinding`	hand your action grammar to LLMUnity so the model cannot leave it
`TextLengthRule`	how long a reply should be, as a phrase the model can read

Author as assets

Type	Why
`AxisCatalogAsset`	which axes your characters have at all
`TraitPresetAsset`	fixed dispositions: the shy one, the blunt one
`PerceptionWeights`	how much each factor of attention counts, per scene
`SamplingProfile`	temperature, penalties, reply length
`PresetCatalog`	which model and voice a language and tier resolve to
`KeyValueStateStore`	a ready `IStateStore` over any key-value storage you already have
`ActionTargetBehaviour`	the component that makes a GameObject nameable as a target
`ActionVerb`	one declared verb: its name, its parameters, whether it needs a target
`ActionParams`, `NpcActionEvent`	the arguments an action carries, and the UnityEvent you wire in the inspector

> **Coverage, honestly.** The package has 138 public types and the documentation names about a third of them. The rest are data holders, enums and internals that happen to be public — but if you meet a name here that no page explains, that is a gap in these docs rather than something you are expected to already know.

13. Making a build

The step most likely to surprise you.

Models are excluded by default — every one of them. That default is a feature: six models downloaded while experimenting would otherwise silently become a twenty-gigabyte build, discovered at upload time. Turn on the one you ship, in the Model Manager.

An included model is staged into `Assets/StreamingAssets/OfflineAINPC` for the duration of the build and removed afterwards. That is the only way to get a file into a Unity build, and it is why the plugin does not leave models sitting in `StreamingAssets` the rest of the time.

A build with broken model references is refused before it starts rather than produced dead and discovered by a player.

Piper is yours to ship. It and its voices are external files, not Unity assets, and they must be where the built player expects them. A build that works in the editor and is silent as a player is nearly always this.

Before you send it to anyone

- run it on a machine that is **not** your dev machine, ideally without Unity installed;
- find the **startup verdict** line in the player log;
- ask one question, hear one answer.

---

14. When it does not work

In the order these actually happen:

Symptom	Check first
Never speaks, console clean	the voice's `.onnx.json` — missing one is silent
No startup verdict line	the model file: present, whole, referenced
Compiles, but no dialogue features	LLM for Unity missing, or Unity has not recompiled since it appeared
Push-to-talk does nothing	whisper.unity not installed. Typing still works
Works in editor, silent in build	Piper not beside the player, or the path is wrong
She says "MMaraMara VMara Vey…"	cumulative fed to `Append`, or deltas to `AppendCumulative`
Stopped answering, nothing in console	a leaked `_busy` — check every exit path
Everything is very slow	which tier resolved? Below the lowest it runs on the CPU

Troubleshooting maps real error strings to fixes.

---

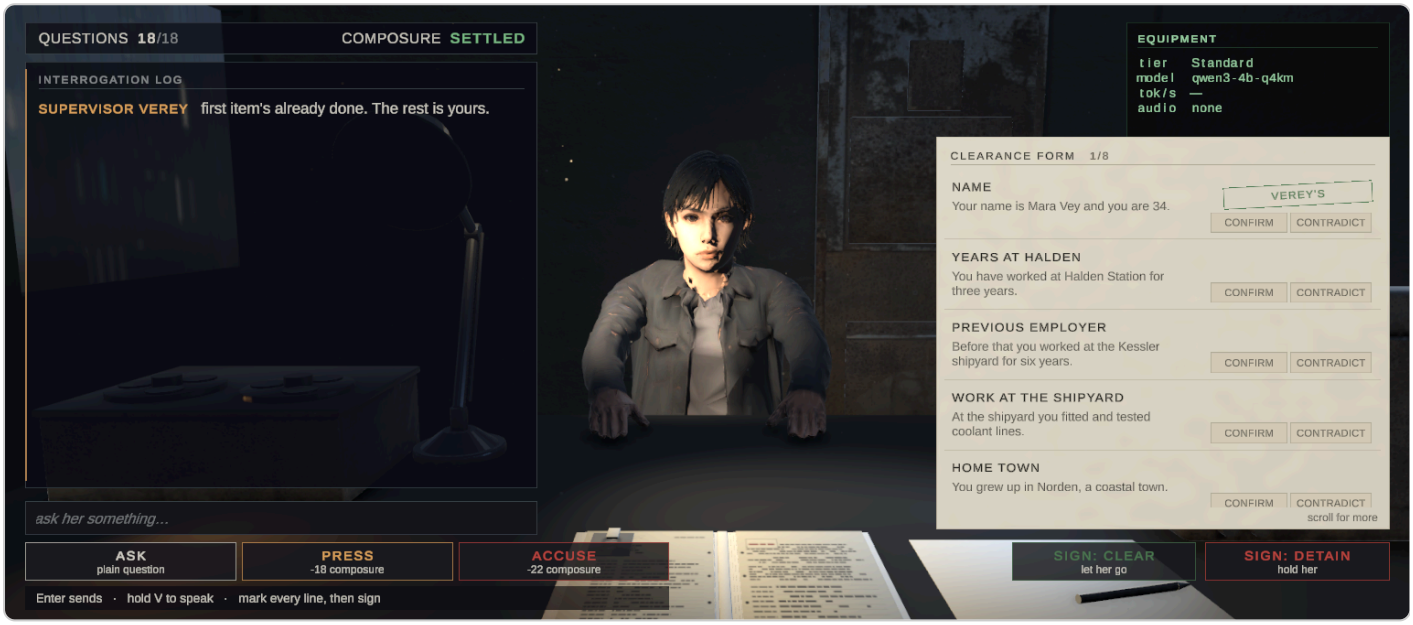
15. Words you will meet

Model / gguf	the language model file. Bigger is smarter and slower
Tier	what your hardware can carry. Decided at startup and printed
Piper / VOICEVOX	the programs that turn text into audio
Whisper	the program that turns the player's speech into text
StreamingAssets	the Unity folder whose contents survive into a build as real files
Prompt cache	why the first reply is slow and the rest are not
Quantisation (`Q4_K_M`)	how heavily the model was compressed to fit in memory. Smaller file, slightly worse answers. `Q4_K_M` is the usual balance
Parameters (`4B`)	the model's size in billions of weights. 4B is the sweet spot for a game; 7B needs a bigger card
GPU layers (`ngl 36`)	how much of the model sits on the graphics card. All of it is fastest; the rest runs on the CPU
Context window	how much text the model can hold at once — system prompt, memory replay and the current turn together
System prompt	the standing instructions the character never says out loud: who she is, what language to answer in, what she can see
Token	a piece of a word. Models are billed and limited in these, not in characters
Streaming	the reply arriving piece by piece, so speech can start before the sentence is finished
Verb / action	a decision the character can make, in a form your code can execute. You invent them — see Actions
Grammar	the rule that makes an action structurally impossible to malform, because invalid words are removed from the model's choices
Axis / band	a number describing a character (`trust: 7`) and the named range it falls in (`warm`). The model is told the band, never the number
Point of interest	an object marked as noticeable, so she can mention what is actually there instead of inventing a fireplace
Speaker id	which voice inside an engine to use. One Piper voice per process; VOICEVOX numbers its characters

Utterance gate	the check that drops a silent or hallucinated recording before it costs a turn
Push-to-talk	holding a key to record. The alternative — always listening — transcribes the room

The interview demo

A five-minute scene where you question a woman across a desk and decide whether to clear her. She is not on rails: every answer comes from a language model running on your machine, she hears you through your microphone, and she speaks with a voice synthesised locally.



> **Play it first, if you would rather see than read:** https://merrymaker14.itch.io/the-interview — a free Windows build, about 788 MB, which fetches its language model on the first launch.
>
> **Where it lives.** The scene is `Assets/Demo/Scenes/InterviewDemo.unity` in the project repository. It is not packaged into `Samples~` yet — the three samples that ship are HelloNpc, LivingCharacter and WorldAware. This page describes it because it is the fullest answer to "what does a finished thing built on this look like".

What it is for

Three questions get asked about a local-model plugin, and a demo is the only honest way to answer any of them:

The question	What the demo shows
Does it actually work?	A whole conversation, spoken aloud, with nothing authored in advance
Will it run on my machine?	The equipment panel prints your tier, your model, your tokens per second and your time to first audio, live
What do I have to write myself?	Every piece the demo uses is public API — the interview logic is ordinary C# on top of it

It is deliberately not a technology showcase with buttons labelled "test perception". It is a small game, because a plugin that only demos well is a plugin nobody finishes shipping with.

The premise

You are a clearance officer on a night shift. Your supervisor, Verey, has left you a candidate and a file about her. Your job is to work through the form, decide whether the file and the woman agree, and sign.

There is a catch, and it is the point: **the evidence cannot tell you what she is.** She is a human or an android on a coin flip, and the contradictions in her file are seeded independently of that. Clearing an android is not a loss. Detaining a human is not a loss either. There is no score, because a score would turn it into "guess whether it is an AI", which is a worse game and a dishonest one.

What you get at the end is the truth beside your own marks: what she actually was, which of her claims really did disagree with the file, and where your reading matched.

What you do

Ask her things. Type into the box and press Enter, or hold **V** and speak. Speech is transcribed locally by Whisper and sent as a question the moment it is recognised — no second keystroke, because having said something out loud and then being asked to press a key breaks the one illusion the demo exists to sell.

Three ways to ask, and they cost different amounts:

	What it does	Composure
ASK	a plain question	**+6** — neutral ground lets her settle
PRESS	lean on an answer that disagrees with the file	**−18**
ACCUSE	say out loud that she is a machine	**−22**

Composure starts at **70** and is visible in the status line as a band: `Settled`, `Guarded`, `Cornered`. It is not decoration — below **8** she stops cooperating and the interview ends whether you were finished or not. About five

hard presses gets there, so pressure is a resource you spend rather than a button you hold.

Repeating a question she has already answered costs 7. Long silences cost 4: she notices.

Mark the form. Eight items, each with the file's version of it. Confirm the ones she backs up, contradict the ones she does not. The first item is already filled in — that is Verey's, and the form says so; override it if you disagree.

Sign. `CLEAR` or `DETAIN`, at any point. You have eighteen questions; signing early is allowed and sometimes the right call.

What to watch while you play

The green panel, top right, is the plugin talking about itself:

```
tier: Standard
model: qwen3-4b-q4km
tok/s: 20.9
to first audio: 2532 ms
```

- **tier** — what the hardware probe decided your machine can carry (Hardware tiers)
- **model** — which model that tier chose from the catalog
- **tok/s** — generation speed on your GPU, measured, not claimed
- **to first audio** — from your question to her first spoken syllable

That last number is the one that matters, and it is measured the honest way: not "the model finished" but "sound came out of the speakers". The first turn of a session is slower than the rest — the prompt cache is filling — and Limitations gives the distribution across twenty-one measured turns.

Watch her, too. She breathes, shifts her weight, and turns her head — body first, then eyes — toward things you mention that are actually in the room. The tape recorder is a real point of interest; she looks at it when it comes up, and she does not invent objects that are not there.

What is plugin and what is demo

Worth knowing if you are reading the code to copy from it.

Comes from the plugin	Written for the demo
`StreamingReplyParser` — tokens in, speakable sentences out	`InterviewSession` — turns, composure, endings
`StreamingSpeechSession` — synthesis and playback in order	`CandidateFactory` — the woman, her file, the seeded contradictions
`PersistentMicRecorder`, `UtteranceGate` — push-to-talk and the silence filter	`InterviewHud` — the desk, built in code
`TTSEngineManager`, `PiperEngine` — the voice	`CandidateGaze`, `SeatedIdle` — where she looks and how she sits
`AIPointOfInterest`, `PerceptionQuery` — what she can see	`FaceAnimator` — blinking and mouth frames
`TierCascade`, `ModelCatalog` — what runs here	`InterviewRoom` — the glue between them

The demo assembly references the plugin and nothing of the game around it. That is the interesting property: it was written against the public API by someone with the source open, which is how several of the documentation pages you are reading got corrected.

Running it

1. Open `Assets/Demo/Scenes/InterviewDemo.unity`.
2. Press Play. The first start loads the model — several seconds during which the equipment panel is still filling in.
3. Ask her something.

It needs the same three things the rest of the plugin needs: a model in `StreamingAssets`, Piper with a paired voice, and — for speech input only — a Whisper model. `Tools ▸ Offline AI NPC ▸ Check my setup` names whichever is missing.

English only, and the settings screen says so rather than pretending. The demo ships one voice, and her persona, the file and the epilogues are written in English. The language row lists all four the plugin has configured and greys out the three this download cannot voice — they need a voice file the demo does not carry. Dialogue and recognition would work in any of them; the pipeline underneath is the same one that speaks Japanese and Spanish (Languages).

Ask her in another language and she declines rather than answering: an English synthesiser handed Russian produces phonetic noise, and a player hearing that concludes the speech is broken rather than the language unsupported.

Desktop only, in this version. The scene switches XR off on purpose and runs on mouse and keyboard. PC VR is planned — the mode detection is one line, but the HUD is a screen-space canvas that does not exist inside a headset, so a VR build needs it in world space with ray-based input first. Standalone Quest is not a target and will not become one: the model is gigabytes and the speech engines are separate desktop processes.

The endings

Four, from what she was crossed with what you signed, and none of them is a verdict on you:

	You cleared her	**You detained her**
She was human	she goes back to her shift	her sister keeps writing to the station
She was an android	it holds, for now	the file that came down afterwards said she was human

The reason to play it twice is that the other assignment reads completely differently against the same evidence.

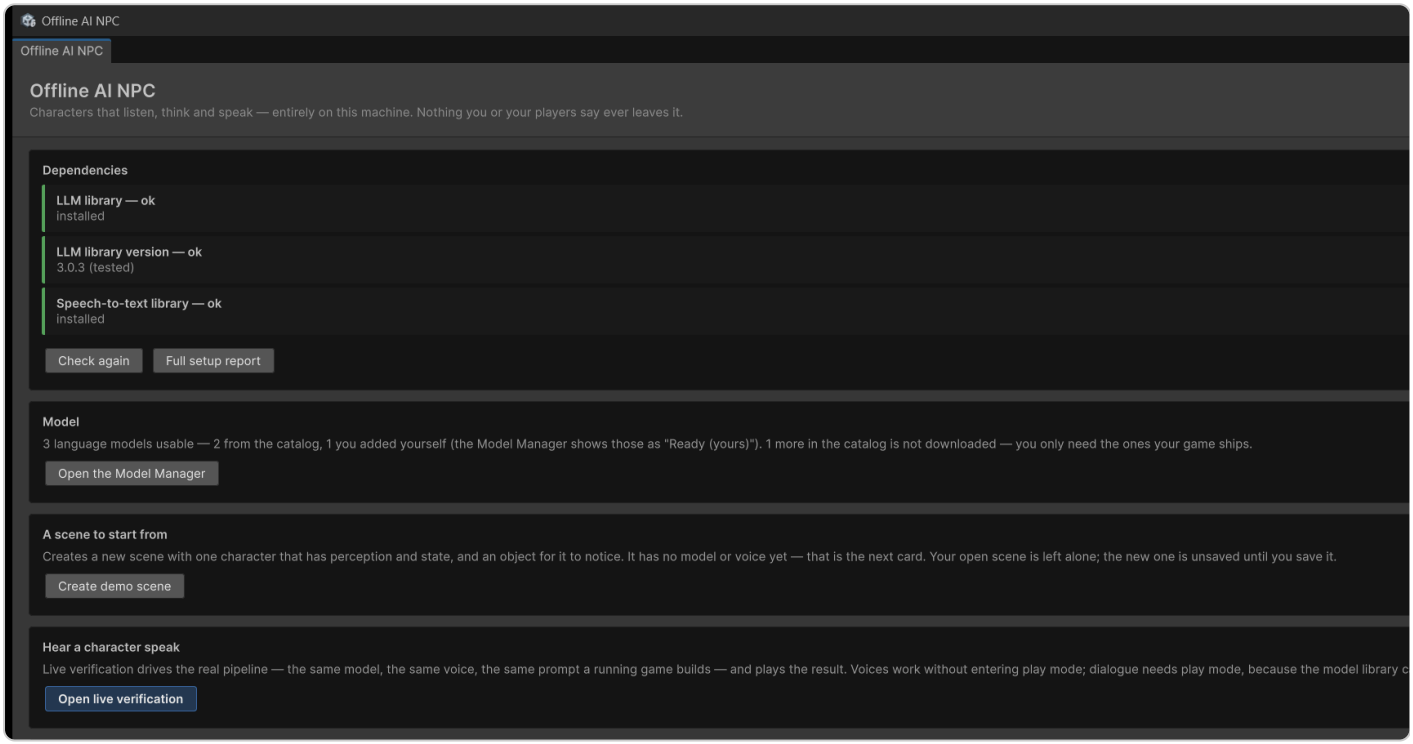
Getting started

From import to a character who answers out loud. Read Is this for your game? first if you have not — it takes two minutes and it is the page most likely to save you a refund.

---

0. The window that opens by itself

The first editor load after import opens **Welcome** — `Tools ▸ Offline AI NPC ▸ Welcome` if you want it back. It runs the checks on this page for you and links to the two windows below.



It shows **once**, on the first load, and never reappears — not after a recompile, not in the next project. It changes nothing in your project on its own: no Project Settings, no scene edits, no files. Everything it can do is behind a button you press.

Three cards, in the order the questions come up: are the dependencies installed, is there a model to talk to, and a scene to start from. If you would rather work from the pages, close it — nothing later depends on having used it.

---

1. Dependencies

Two, both external and free, neither bundled:

	Package id	Needed for
LLM for Unity	`ai.undream.llm`	the language model. Required
whisper.unity	`com.whisper.unity`	speech input. Optional — typed input works without it

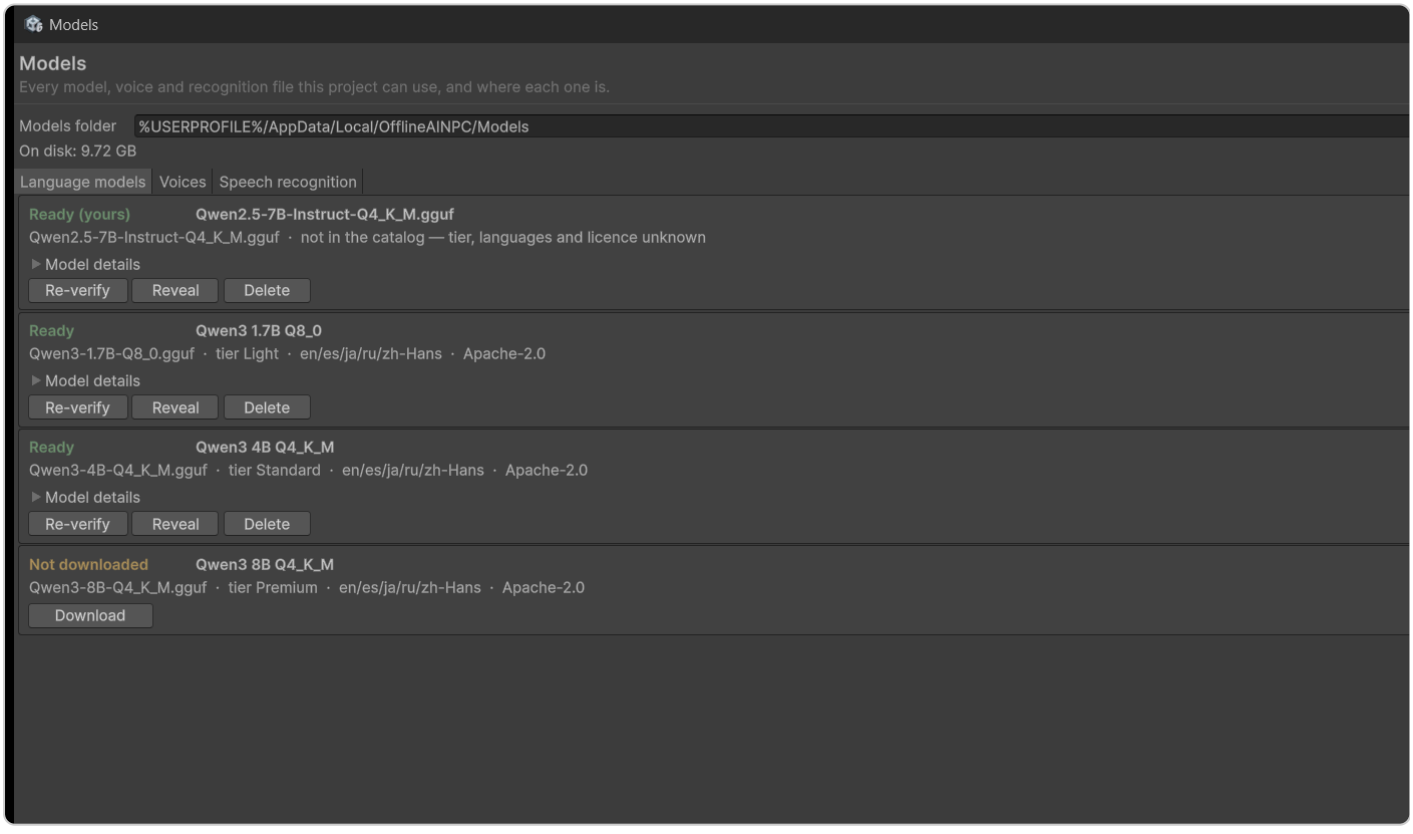
The plugin compiles without either: the features that need them are compiled out through `versionDefines`. A missing dependency is a missing feature, never a compile error.

And there is no third one. Nothing else is required — not Newtonsoft.Json, which comparable plugins ask for and which this one deliberately does without. The reason is what a clean import looked like while it did depend on it: 36 compile errors before a single window could be opened, in any project that had not installed Newtonsoft for other reasons. The plugin reads its own JSON now (`OfflineAINPC.Core.Serialization`), so `Assets/OfflineAINPC` compiles in an empty project with nothing installed at all — verified by importing it into one.

LLM for Unity does need Newtonsoft, and declares it: installed as a package it arrives on its own, installed as an Asset Store folder you add `com.unity.nuget.newtonsoft-json` yourself. Either way that is its requirement, met once, and not something this package drags in on top.

2. A model

`Tools ▸ Offline AI NPC ▸ Model Manager`.



Each **catalog** entry states its tier, its languages and its licence before you download anything. `[Ready]` means the file is on disk and its header reads correctly — not merely that the download finished.

A file you dropped in yourself shows as `[Manual]`: usable, but the manager knows nothing about it beyond what it can read from the file, so the row says so instead of leaving the three fields blank. The Welcome window counts those separately for the same reason — "3 models ready" and "1 [Ready], 2 [Manual]" were the same three files described by two windows that disagreed.

Pick the preset for your language and tier and download it. The manager verifies the file against a hash, so a truncated download is caught before it is loaded rather than as a mysterious failure later. Presets carry their licence, and the default line-up is chosen to need no attribution in your shipping game.

If you already have a `.gguf`, put it in `StreamingAssets` and name it as the manifest expects.

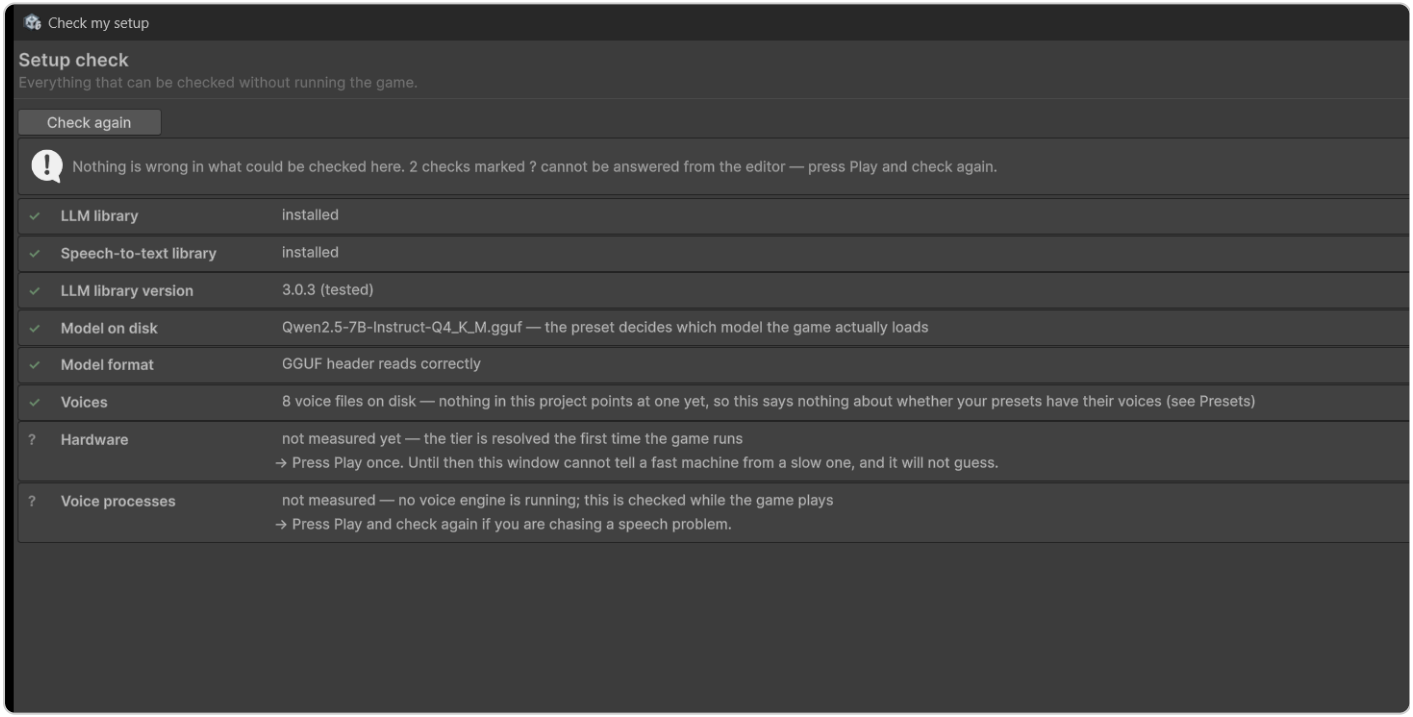
3. A voice

Install Piper and put its folder somewhere your build can reach. Add at least one voice — **both files**, `name.onnx` and `name.onnx.json`. A missing `.json` gives you silence with no error, and it is the most common setup problem there is.

For Japanese you supply the engine — the package ships Piper only. VOICEVOX is the usual choice and connecting it is about eighty lines against one interface; the complete worked example is in Languages.

4. Check

`Tools ▸ Offline AI NPC ▸ Check my setup`.



Every line is a real check against your project, and a failing one says what to do rather than what went wrong. The example above is the most common failure there is: a voice the configuration points at is not on disk, which produces **silence with no error**.

It reports what is present, what is missing, and what to do about it — and where it cannot determine something, it says so instead of guessing. Fix anything red before continuing; every red item is something that would otherwise surface later as odd behaviour.

5. A character

Add to a GameObject in your scene:

- an `AudioSource` for her voice,
- your dialogue controller — copy the one in Dialogue (there is no runnable sample yet),
- optionally `NpcStateComponent` and `PerceptionQuery` once she works.

Give her a persona: who she is, how she speaks, a handful of facts. Keep it short. Everything in the system prompt is re-read on every turn, and a small model does better with five clear facts than with two paragraphs.

6. Speech recognition

Push-to-talk: start recording on key down, stop on key up, hand the transcript to your controller.

Two things that are not obvious and will cost you an evening each:

Silence is transcribed as words. Whisper hallucinates on an empty recording — "Thank you for watching" and similar. Gate on RMS and drop anything below a threshold, and filter the known phrases. Without this, a stray key press starts a conversation.

Any word can be a trigger. If you route on keywords, use word boundaries. A hallucinated fragment containing your trigger word inside another word will fire it.

7. Run it

Enter Play mode, hold the key, ask her something.

The first reply of a session takes about 2.1 s while the prompt cache fills; after that, around 2.0 s to the first word on the reference machine. If your first turn is slower than that, it is normal. If your tenth is, see Troubleshooting.

Adding your own assembly

If your game code lives in its own assembly definition, reference `OfflineAINPC.Runtime`:

```
{
    "name": "MyGame.Characters",
    "references": [ "OfflineAINPC.Runtime" ],
    "autoReferenced": true
}
```

For a test assembly alongside it:

```
{
    "name": "MyGame.Characters.Tests",
    "references": [ "MyGame.Characters", "OfflineAINPC.Runtime",
                    "UnityEngine.TestRunner", "UnityEditor.TestRunner" ],
    "includePlatforms": [ "Editor" ],
    "overrideReferences": true,
    "precompiledReferences": [ "nunit.framework.dll" ],
    "defineConstraints": [ "UNITY_INCLUDE_TESTS" ],
    "autoReferenced": false
}
```

`OfflineAINPC.Runtime` references **nothing but the engine** — not your game, not LLMUnity, not whisper.unity, and no third-party library. That is what makes it portable between projects, and it is why an import cannot fail on something you have not installed.

Where to go next

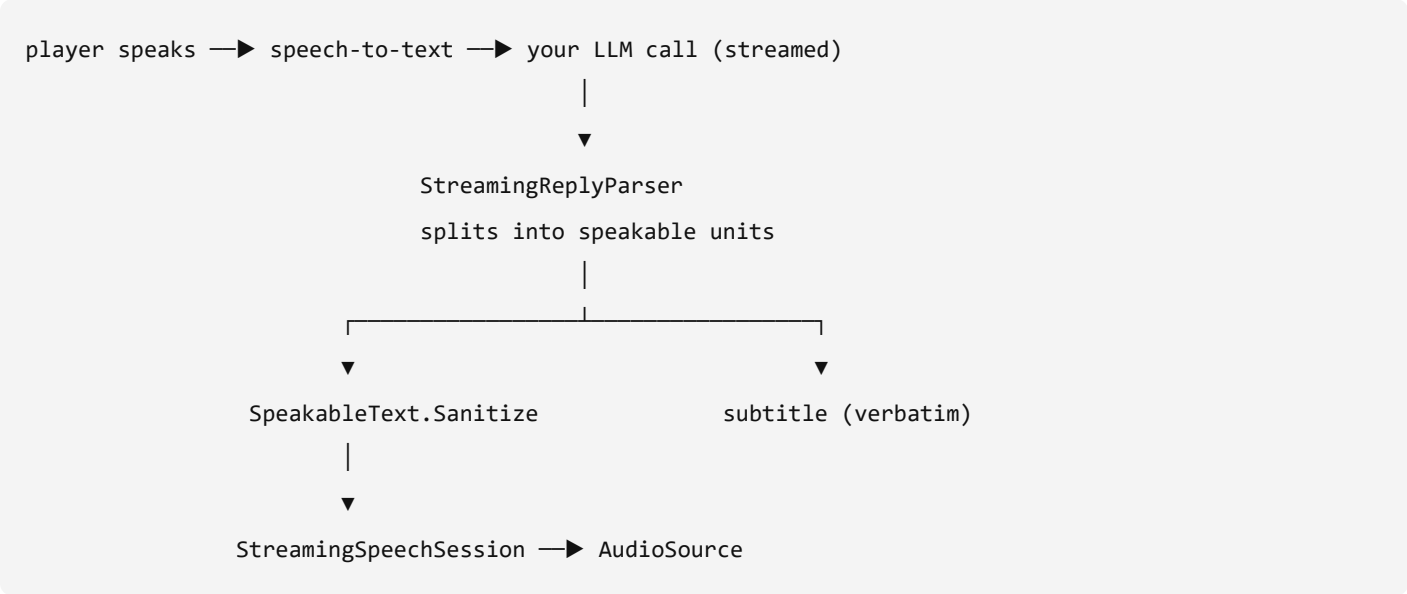
You want	Page
A character who notices the room	Perception
Moods, needs, relationships	State — start at the short path
A character who *does* things	Actions
Another language	Languages
To know what will bite you	Limitations

Dialogue: wiring a talking character

This is the page that turns the parts into a character: the fifty lines that sit between a microphone and a voice, and what each of them is for.

Everything below is the working controller, not a sketch. Where a line has a trap in it, the trap is marked at the line.

The shape



Two things about this diagram matter more than they look.

The reply is split into sentences and spoken as they arrive. Waiting for the whole reply costs 2.4 s to first audio on the reference machine; speaking from the first finished sentence costs 2.0 s. That is the difference between the two figures quoted everywhere in this documentation, and it is why the parser exists.

The voice and the subtitle get different text. An emoji is fine to read and impossible to speak. `SpeakableText.Sanitize` strips what a synthesiser cannot say — emoji, markdown marks, control characters — and leaves every language's punctuation alone. The subtitle gets the original.

The whole controller

Minus your project's specifics. It compiles against the public API and nothing else.

```
using System;
using System.Collections;
using UnityEngine;
using OfflineAINPC.Core;
using OfflineAINPC.Speech;

public sealed class TalkingCharacter : MonoBehaviour
{
    [SerializeField] private AudioSource _voice;
    [SerializeField] private int _speakerId;          // which voice, when the engine has several

    private StreamingReplyParser _parser;
    private StreamingSpeechSession _speech;
    private bool _busy;

    /// <summary>One turn: the player said something, the character answers.</summary>
    public IEnumerator Say(string playerLine)
    {
        // ONE lock, released on every exit path. Getting this wrong is how push-to-talk
        // stops working forever; a `finally` is not optional here.
        if (_busy) yield break;
        _busy = true;

        try
        {
            _parser = new StreamingReplyParser();
            _speech = new StreamingSpeechSession(
                engine:    () => TTSEngineManager.Instance != null ? TTSEngineManager.Instance.Active :
null,
                audio:     _voice,
                speakerId: _speakerId,
                onSubtitle: (text, seconds) => ShowSubtitle(text, seconds),
                onWarning:  message => Debug.LogWarning(message));

            StartCoroutine(_speech.RunSynthesis());
            StartCoroutine(_speech.RunPlayback());

            // YOUR llm call goes here. Feed what the callback gives you to the parser and
            // hand every finished unit to the speech session.
            //
            // MIND WHICH ONE YOUR BACKEND GIVES YOU. LLMUnity's stream callback is
            // CUMULATIVE — every invocation carries the whole reply so far — so it needs
            // AppendCumulative. A backend that streams true deltas needs Append. Getting
            // this backwards is silent and unmistakable: the character says
```

```
            // "MMaraMara VMara VeyMara Vey, 34." instead of "Mara Vey, 34."
            yield return GenerateReply(playerLine, onToken: textSoFar =>
            {
                _parser.AppendCumulative(textSoFar);       // LLMUnity: cumulative
                // _parser.Append(delta);                  // a backend that streams deltas
                while (_parser.TryDequeueUnit(out var unit)) _speech.Enqueue(unit);
            });

            _parser.Complete();
            while (_parser.TryDequeueUnit(out var unit)) _speech.Enqueue(unit);
            _speech.InputComplete();

            while (!_speech.Finished) yield return null;
        }
        finally
        {
            _busy = false;
        }
    }

    private void ShowSubtitle(string text, float seconds) { /* your UI */ }

    private IEnumerator GenerateReply(string playerLine, Action<string> onToken)
    {
        // LLMUnity, your own binding, or a stub. The plugin does not care which.
        yield break;
    }
}
```

The parts you must not skip

`_busy` released in a `finally`. Every early return, every exception, every cancelled coroutine has to clear it. This project has broken push-to-talk this way more than once: the symptom is that the character simply stops responding, with nothing in the console.

`InputComplete()`. Without it the synthesis loop waits forever for a unit that is never coming, and the character never finishes speaking.

A null engine is normal. The engine reference is a `Func<>` and not a stored reference because the engine is replaced when the language changes. During that switch it is null; the session handles it by delivering the line to the subtitle only. Do not assume it is there.

The prompt cache

The system prompt is cached by the inference engine. A stable prompt means fast replies; a prompt that changes every turn pays the cold cost — about 2.1 s here — every time.

So: build the prompt once, at `Start`, and re-build it only when something real changed — the language, the character's persona, a new ambient fact. Never per turn.

That is also why per-turn context (what the character can currently see, who is nearby) is appended to the *user* message rather than folded into the system prompt: it changes constantly, and putting it in the cached half would throw the cache away every turn.

Memory

`NPCMemory` keeps a rolling window of the conversation, keyed per character, over a store you choose — so it survives scene loads and restarts once you give it one that persists.

The whole of it is three calls. Pick a store on the first line, and never think about persistence again:

```
using OfflineAINPC.Memory;

NPCMemory.Store = new PlayerPrefsStore();        // ships with the plugin; one line, done

NPCMemory.Append(npcId, NPCMemory.RoleUser, playerLine);
NPCMemory.Append(npcId, NPCMemory.RoleNpc,  reply);

string replay = NPCMemory.GetContextString(npcId, lastN: 10);
```

Fold `replay` into the system prompt when you build it. The window is deliberately small: every remembered turn is prompt tokens on every subsequent turn, and a 4B model's attention does not reward a long transcript.

Do that assignment once, at startup, before anything talks. Without a store `NPCMemory` still works and simply forgets on quit — check `NPCMemory.HasStore` if you want to be sure which of the two you have.

Choosing a store

`PlayerPrefsStore` is the right first choice and the wrong last one. It is a small global blob with no save slots and no encryption, which is fine for a demo and wrong for a game that already knows how to save.

Anything else is three methods — `Get`, `Set`, `Delete` — against `IKeyValueStore`. Here is a complete one that writes JSON files under `Application.persistentDataPath`, per save slot:

```
using System.IO;
using OfflineAINPC.Memory;
using UnityEngine;

public sealed class FileStore : IKeyValueStore
{
    private readonly string _dir;

    public FileStore(string slot = "slot0")
    {
        _dir = Path.Combine(Application.persistentDataPath, "npc-memory", slot);
        Directory.CreateDirectory(_dir);
    }

    // Keys arrive as "memory_<npcId>", so they are already safe file names — but a game
    // that invents its own ids should still sanitise, not trust.
    private string PathOf(string key) => Path.Combine(_dir, key + ".json");

    public string Get(string key)
    {
        string path = PathOf(key);
        return File.Exists(path) ? File.ReadAllText(path) : "";
    }

    public void Set(string key, string value) => File.WriteAllText(PathOf(key), value);

    public void Delete(string key)
    {
        string path = PathOf(key);
        if (File.Exists(path)) File.Delete(path);
    }
}
```

`NPCMemory.Store = new FileStore(currentSlot);` and memory follows the save slot. The same three methods reach a database, a cloud save, or the dictionary a test uses:

```
// In tests: no files, no PlayerPrefs, nothing left behind.
public sealed class MemoryStore : IKeyValueStore
{
    private readonly Dictionary<string, string> _map = new Dictionary<string, string>();
    public string Get(string key) => _map.TryGetValue(key, out var v) ? v : "";
    public void Set(string key, string value) => _map[key] = value;
    public void Delete(string key) => _map.Remove(key);
}
```

The store is asked for a value once per character per turn, not per frame, so a store that touches disk is not a performance decision.

What it keeps, and what it costs

`Append(npcId, role, text)`	adds one line; roles are `RoleUser`, `RoleNpc`, `RoleOther`, `RoleSummary`
`GetContextString(npcId, lastN)`	the last `lastN` **exchanges**, formatted for a prompt
`Clear(npcId)`	forgets that character, store included
`MaxEntries`	40 lines per character — older lines fall off the end
`Compactor`	optional: your own summariser, called when the window overflows

Two lines per exchange, forty entries: about twenty turns per character before the oldest starts dropping. Raise it and every turn costs more prompt tokens; that is the whole trade-off, and it is why the default is a number rather than "everything".

Voices

The language decides the default voice (Languages). A character overrides it by speaker id — that is how two English speakers in one scene sound different. With Piper, each voice is a separate process, spawned lazily; with VOICEVOX, a speaker id on one server.

Spawn the voices you will need up front rather than on the first line. A Piper process takes about 1.3 s to come up, and paying that mid-scene is an audible silence.

What this page does not cover

- **Speech recognition** — see GettingStarted.

- **Actions** (the character doing rather than saying) — see Actions.
- **Perception** (the character knowing what is around her) — see Perception.
- **Lip sync.** Not included; the demo project drives a jaw bone itself.

Internal state

> **What this does, honestly.** State *colours* what a character says; it does not make her act. Measured on Qwen3-4B: asked directly how she is at `sleepiness 95/100`, she referred to being tired in **6 of 10** replies, against **0 of 10** with the state system switched off. On a relationship axis the same probe scored **6 of 10** against a control of **1 of 10**. It is a real, reproducible effect and a probabilistic one — expect a character who often mentions how she feels when the subject comes up, not one who always does, and not one who volunteers it unprompted in a one-sentence reply.
>
> The deterministic half is not probabilistic at all: threshold events, trait numbers, save/load, and the model being structurally unable to change a need all behave exactly as specified, every time.

The short path first

Most projects need one axis that a few game events move, with a visible consequence they control themselves. That does not need an asset, a trait preset, persistence or a tick. **It needs this:**

```
using OfflineAINPC.State;

// One axis, in code. No asset, no inspector.
var composure = new AxisDefinition
{
    Id = "composure", Kind = AxisKind.Mood,
    Min = 0f, Max = 100f, Baseline = 70f,
    Bands =
    {
        new AxisBand { Id = "cornered", Min = 0f,  Max = 29.99f },
        new AxisBand { Id = "guarded",  Min = 30f, Max = 64.99f },
        new AxisBand { Id = "settled",  Min = 65f, Max = 100f },
    },
};

var state = new NpcStateModel("suspect", new AxisCatalog(new[] { composure }));

// A game event moves it. Deterministic, exact, every time.
state.Modify("composure", -18f, reason: "pressed on a contradiction");

// And you read the band to decide what the player SEES.
string band = state.BandOf("composure")?.Id;      // "guarded"
```

That is the whole system for that case. `NpcStateModel` is plain C# — no `MonoBehaviour`, no scene, no Unity — so it can be unit-tested directly.

Reach for the rest of this page when you want descriptor text reaching the model, several characters with different dispositions, decay over real time, persistence across sessions, or the model being allowed to move an axis itself. Each of those is a reason for one more piece of the apparatus below. If none of them applies, you are done.

The full apparatus

Four axis kinds. They are separate because they live on different timescales, and merging any two of them breaks one of them.

Kind	Lifetime	Persisted	Ticks
Mood	minutes, eases back to a baseline	no	toward baseline
Relationship	permanent, per pair	yes	never
Need	grows until satisfied	yes	away from baseline
Trait	never changes	yes	never — it modulates the rest

Put a mood and a need in one bucket and either the mood stops decaying or the need starts. That is not a hypothetical; it is the usual version of this bug.

Defining an axis

Create an **Axis Catalog** asset. Each axis carries its range, baseline, rate per hour, hysteresis, how far the model may push it, and its descriptor bands.

```
trust   0..20, baseline 0, no decay, model may move it ±0.5
  guarded  0..2.99   "is polite but a little guarded with you"   (always mentioned)
  warm     3..5.99   "is comfortable around you"
  close    6..9.99   "trusts you and lets it show"
  devoted 10..20     "trusts you completely"
```

Never send a number to the model. Given `trust: 7` a small model either ignores it or plays it as a crisis. Bands exist so a designer writes the words once.

Band text is per language, through the same `LocalizedText` the perception descriptions use. A sixth language is data.

Reading and writing

```
var state = GetComponent<NpcStateComponent>();

state.Get("trust", "player");                       // read
state.Modify("sleepiness", 5, "carried her upstairs");
state.Modify("trust", 1, "brought tea", "player");  // relationships are per pair

state.ThresholdCrossed += c =>                      // once per crossing, with hysteresis
    Debug.Log($"{c.AxisId}: {c.FromBandId} -> {c.ToBandId}");
```

Ask in **bands** rather than in numbers wherever gameplay is deciding something:

```
if (state.AtLeastBand("trust", "warm", "player") ?? false) OfferTheHug();
```

`AtLeastBand` returns `bool?`, and the third answer is the point. Null means the question could not be answered — no such axis, no such band, or a value the bands do not cover — and it is deliberately not `false`. A caller that cannot tell "I do not know" from "no" gates content on a typo and never finds out.

The reason to prefer it: a literal `trust >= 3` in your code is a copy of a number in the asset, and the copy does not move when you retune. Retune the asset and the character *describes* herself differently while *behaving* exactly as before.

The threshold event never fires on load: progress made last session is not progress made now, and replaying every milestone at every launch is how a celebration stops meaning anything.

Traits

A trait shifts an axis baseline, scales its rate, and scales how hard incoming changes land. Two characters with identical axes then behave differently — without it they all converge on the same emotional shape and the axes stop meaning anything.

```
shy:        mood baseline −2, mood rate ×0.7, trust sensitivity ×1.3, sleepiness rate ×1.3
energetic:  mood baseline +2, mood rate ×1.4, trust sensitivity ×0.7, sleepiness rate ×0.8
```

What the model may do

Add a **Feel Action Handler** and the `feel` verb appears, with `axis` as a grammar choice built from the axes that opened themselves to it:

```
{"verb":"feel","params":{"axis":"mood_valence","delta":-2,"reason":"..."}}
```

Axes with `MaxModelDelta = 0` are not in that choice, so a need is **unreachable** rather than rejected — a character cannot decide she is no longer tired. Open axes are clamped per reply.

Gating verbs by state

```
NpcStateComponent.VerbGate.RequireBand("race", "sleepiness", maxBand: "drowsy",
                                       note: "too tired to run");
```

The verb leaves the grammar. She does not refuse it — it was never offered. Verbs are filtered by state, targets by perception, and neither knows about the other.

`Require(..., min:, max:)` takes numbers and still works. Prefer `RequireBand`: a band bound resolves against the catalog every time the rule is evaluated, so moving the band's edge moves the rule with it. A numeric bound is a second tuning of the same thing, and the two drift. Put `max: 85` on a rule while the `spent` band begins at 70, and for those fifteen points a character is described as barely keeping her eyes open while still being offered the chance to look around.

A band the catalog does not define is **ignored loudly**, once, in the console. Enforcing a typo would remove a verb forever; ignoring it quietly would make the gate a check that cannot fail.

Saving

Persisted axes serialise as `{schemaVersion, values}` through `IStateStore`. A project with its own save format implements the interface, or uses `Export()` / `Import()`:

```
Dictionary<string, float> mine = state.Export();   // fold into your own save
state.Import(mine);                                // restore, raising no milestones
```

Mood is session-only by design.

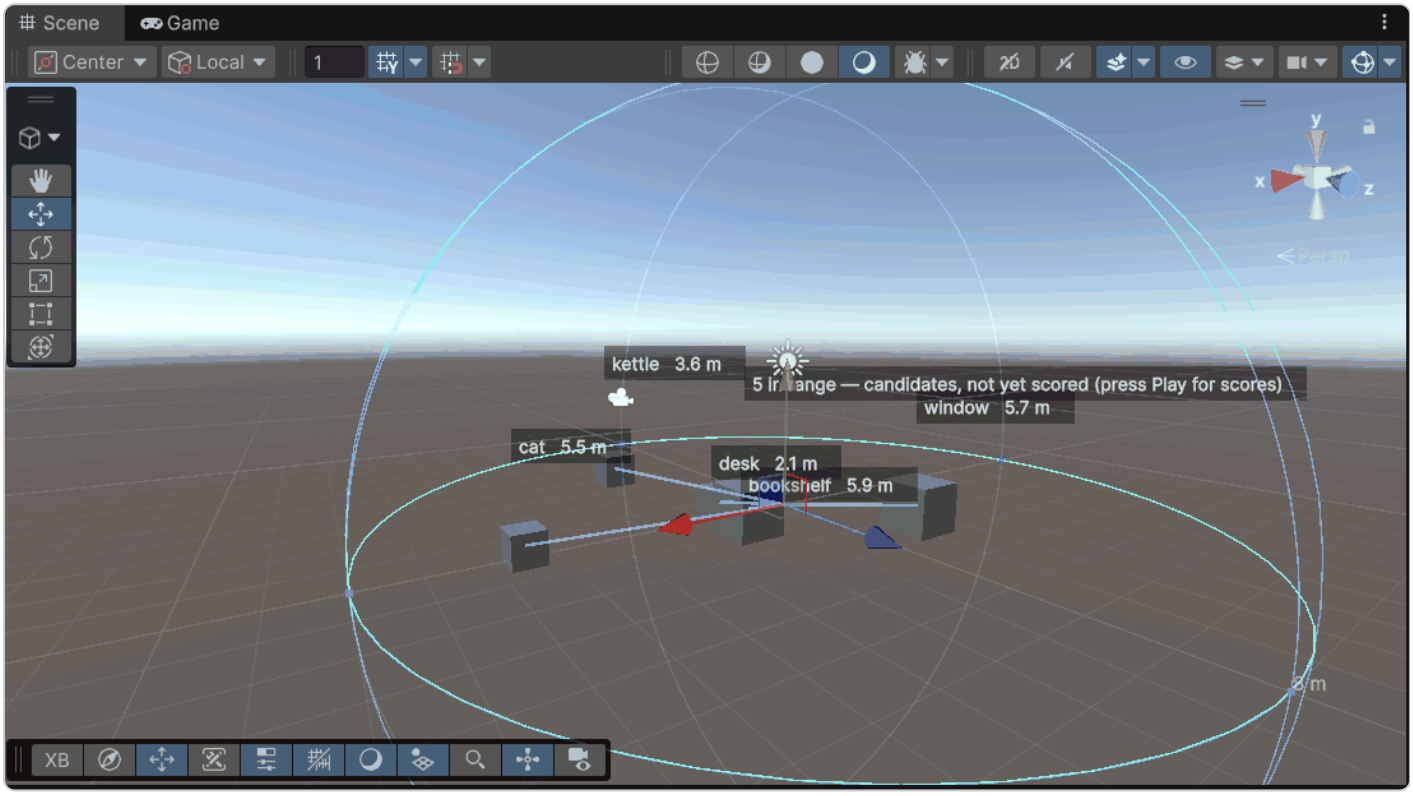
Pause behaviour

Ticks accumulate scaled `Time.deltaTime`, so at `timeScale = 0` nothing accumulates and nothing ticks — she does not get sleepier while the player is in a menu, and no timer spins. Perception freezes with it, for the same reason: both model game time.

The speech queue deliberately does the opposite and keeps running on unscaled time, because `AudioSource` ignores `timeScale` and a line already sounding does not stop. See Limitations for the full per-subsystem table.

Points of interest

Make an object noticeable: add **AI Point Of Interest** to it. That is the whole setup.



Select a character and the radius is a handle you can drag. Everything inside it is a candidate; what reaches the prompt is the highest-priority handful, recomputed per turn.

The scene view answers two different questions depending on whether the game is running:

- **Not running** — every point of interest inside the radius, named, with its distance. These are candidates and they are deliberately not scored: distance is knowable while you are placing furniture, line of sight and priority are not, and a number invented in edit mode would be a guess wearing the clothes of a measurement.
- **Playing** — the objects actually in context, each with the score that put it there, the line thicker for the ones in view. This is the answer to "why is she talking about the bookshelf and not the bed".

There is no list to register it in, no manager to tell. The component adds itself to a static registry while it is enabled and removes itself when it is disabled or destroyed, so object pooling, additive scene loads and scene unloads are already correct.

The object

Field	What it does
`id`	What the model writes to name it. Short, unique, obvious. Derived from the object name if left empty.
`description`	What it is, per language.
`priority`	Higher is more worth mentioning. 0 is ordinary scenery.
`affordances`	Verb ids this object accepts. Empty means all of them.
`states`	Authored states selected at runtime: `poi.SetState("open")`.
`On Npc Interact`	UnityEvent raised when an NPC acts on it — inspector wiring, no code.

Ids matter more than they look. The model picks a target by writing its id, so two objects called `chair` means it cannot reliably mean either. Duplicates and over-long ids are warned about in `OnValidate`.

A point of interest is also an action target. It implements `IActionTarget` and `IAimTarget`, so a verb can be aimed at it without adding an Action Target component as well — see Actions. The reverse does not hold: an Action Target is addressable but not perceived, so the character can be told to act on it and will never mention it unprompted.

The character

Add **Perception Query** to the NPC.

```
score = w·proximity + w·priority + w·facing + w·recency-of-interaction
```

Weights are serialized data. Two rules are not negotiable:

- **Occlusion excludes.** Something behind a wall is not low-scoring, it is absent. A character remarking on an object she cannot see reads as broken no matter how the weights are tuned.
- **Only the top N reach the prompt** (default 6, hard cap 15). Small models degrade quickly once a prompt turns into a list, so this is a quality control.

Facing is soft by contrast — something behind her is less salient, not invisible, because she can turn her head.

How it reaches the model

The result of one recompute produces two things at once:

```
Nearby: a mug of cocoa (still warm) [mug], the record player (playing) [record_player].
```

…and the target enum of the action grammar, built from that same list. So the character is **physically unable** to name an object it cannot currently see, and what it is told about can never disagree with what it may refer to.

The block rides with the turn and is never written to the conversation transcript: it describes one moment, and a transcript would repeat it later as though it were still true.

Receiving an interaction

```
public class Kettle : MonoBehaviour
{
    private void OnNpcInteract(NpcActionContext ctx)   // wired on the POI in the inspector
    {
        Debug.Log($"{ctx.SpeakerId} did {ctx.Verb} to {ctx.Poi.Id}");
    }
}
```

If the object vanished between generation and dispatch, nothing is raised and nothing throws.

Making a whole scene visible

```
PoiBulkAttach.AttachToLayer("Props");     // or AttachToTag("Interactable")
```

Idempotent, and ids are de-duplicated automatically.

Debugging

- Select the NPC: the perception radius and lines to everything currently in context.
- `perception.Explain(poi)` answers "why did she not mention the mug?" with the actual reason — too far, behind which object, or simply outranked.

Structured actions

An NPC reply can carry two things at once: natural speech for the voice, and a machine-reliable decision your game can execute.

```
{"verb":"look_at","target":"window"}
It stopped raining. Come and see.
```

The first line is generated under a **grammar**, so it cannot be malformed. The rest is ordinary prose and flows through the streaming speech pipeline sentence by sentence.

Do not hope the model formats correctly — constrain it. llama.cpp masks invalid tokens at every step, which makes a broken action block structurally impossible rather than unlikely.

What a verb is

A verb is a word you invent, and the plugin ships none of them. `look_at`, `emote`, `open` in these examples are not built in — they exist because somebody wrote a `Register` line for them, the way `"jump"` exists in your input map because you added it.

Three parts, and only the first is required:

Part	What it is	Example
verb	the name of a thing the character can decide to do	`look_at`
target	*what* she decided to do it to, chosen from objects that exist right now	`window`
parameters	the details of the decision	`kind: "shy"`

A verb is not behaviour. Registering `open` does not open anything and does not teach the model what opening means; it does four things:

1. puts the word into the **grammar**, so the model may emit it and may not emit anything else;
2. shows your one-line description to the model, which is all it knows about when to choose it;
3. makes the parser accept `{"verb":"open", …}` and reject a target or parameter the verb never declared;
4. gives the dispatcher something to route to **your** handler.

The behaviour is the handler you write. Nothing happens until you write it — a registered verb with no handler is a decision the character can make and the game will ignore, which logs and costs nothing else.

Register only what your game can carry out. Every verb in the grammar is a thing the model will eventually choose, and a character who decides to do something no code implements is worse than one who never had the option.

Register a verb

```
using OfflineAINPC.Actions;

NpcActions.Register("look_at", "look at something you mention", TargetRequirement.Required);
NpcActions.Register("emote", "let a feeling show", TargetRequirement.None,
                    ActionParameter.Choice("kind", new[] { "happy", "shy", "sad" }));
```

Prefer `Choice` over `Text` wherever the set is knowable: a choice becomes a closed alternation in the grammar, so the model cannot invent a value you have no code for.

Register a target

Add an **Action Target** component to the object and give it a short id, or register one directly:

```
NpcActions.Targets.Register("window", "the tall window by the beds");
```

A disabled target leaves the grammar, so the model literally cannot refer to an object that is not there.

If the object already has an AI Point Of Interest, it is already a target. Perception's component implements the same `IActionTarget`, so adding an Action Target beside it registers one object twice under two ids — and the model picks between them by writing an id. Choose one: Action Target for something a verb can be aimed at, AI Point Of Interest for something the character should also be able to *notice* — see Perception.

If the object already has an AI Point Of Interest, it is already a target. Perception's component implements the same `IActionTarget`, so adding an Action Target beside it registers one object twice under two ids — and the model picks between them by writing an id. Choose one: Action Target for something a verb can be aimed at, AI Point Of Interest for something the character should also be able to *notice* — see Perception.

Receive an action

Add an **Action Dispatcher** to the character. Three ways in:

```
// 1. UnityEvent — wire it in the inspector, no code.

// 2. Code, one verb:
dispatcher.AddHandler("look_at", a => { var at = a.Target.AimPointOf(); if (at) transform.LookAt(at);
});

// 3. Code, an interface:
public class Doors : MonoBehaviour, IActionHandler
{
    public IEnumerable<string> HandledVerbs => new[] { "open" };
    public void Handle(NpcAction action) => Open(action.Target);   // already resolved
}
```

`NpcAction` carries the resolved target object, not just its id, and parameters already validated against the verb. There is nothing left to re-check.

A handler that throws is contained and logged: it loses its action, never the conversation.

Guarantees

- The action block is **optional** — most replies are pure conversation, and the grammar never forces one.
- Speech is **never** empty and never starts with `{`, so an action can never arrive instead of a reply.
- A malformed block degrades into speech rather than swallowing the line.
- A rejected action gets **one** corrective round, then is dropped silently.

Backends

The core does not reference any LLM library. Grammar reaches the backend through `IGrammarSink`; the adapter for LLMUnity ships in the optional `OfflineAINPC.LLMUnity` assembly, which compiles only when LLMUnity is installed. Game code finds it through the interface:

```
var sink = GetComponent<IGrammarSink>();     // null when no backend is installed
```

Note for backend authors: in llama.cpp the grammar is sampler state on the client, not a per-request argument. It persists until cleared, so whatever sets it must clear it.

Which languages does this support?

There is no single honest number, so this page does not print one. A language needs three things to work — a model that speaks it, recognition that hears it, and a voice that says it — and they do not all cover the same set.

The short version: your characters can hold a conversation in roughly a hundred languages. Voices are bundled for a few. For the rest, you supply a voice file, and doing that takes one asset and no code.

Tier 1 — ready out of the box

Language	Voice	Licence	Provenance
English	`en_US-ljspeech-medium` / `-high`	**public domain**	trained from scratch on LJ Speech

English is genuinely ready: a voice, no attribution owed, nothing to clear. If you want many distinct English voices, `en_US-libritts-high` carries 904 of them under CC BY 4.0 — see Licensing for the attribution string and the auditioning tool.

Japanese works, and **you supply the engine**. The package ships one speech engine, Piper, and Piper has no Japanese voice with clearable provenance. The usual answer for Japanese is VOICEVOX — a separate application with an HTTP server — and connecting it is about eighty lines against one interface. The complete file is printed below: a working `ITTSEngine`, not a sketch. The samples in this package speak English through Piper; Japanese is the case you add.

Each VOICEVOX character voice carries usage terms binding the *end user*, not you, and they are **per character** — see Bring your own engine for what to check.

Spanish, Russian, Chinese and everything else are Tier 2 below — not because the model or the recognition fall short, but because no voice with a traceable commercial provenance was found for them. Most published Piper voices are fine-tuned from a research-only base; Licensing explains the test to apply, and it is a test you can apply yourself to any candidate.

Tier 2 — works, bring your own voice

This is the large and interesting tier, and it is where most projects will land.

The language model handles well over a hundred languages and speech recognition covers around a hundred. What is missing for these is only a bundled voice — and pointing the plugin at a voice you supply is a single asset with no code.

Quality across that tail varies a great deal. Korean, German, French, Portuguese, Italian, Chinese, Polish, Dutch and Turkish are all well covered by both model and recognition; a language with little presence in the training data will be understood but spoken awkwardly. Test yours before committing to it: `Tools > Offline AI NPC > Live verification` will speak a line and run a turn in the language you are considering.

Adding a language, start to finish

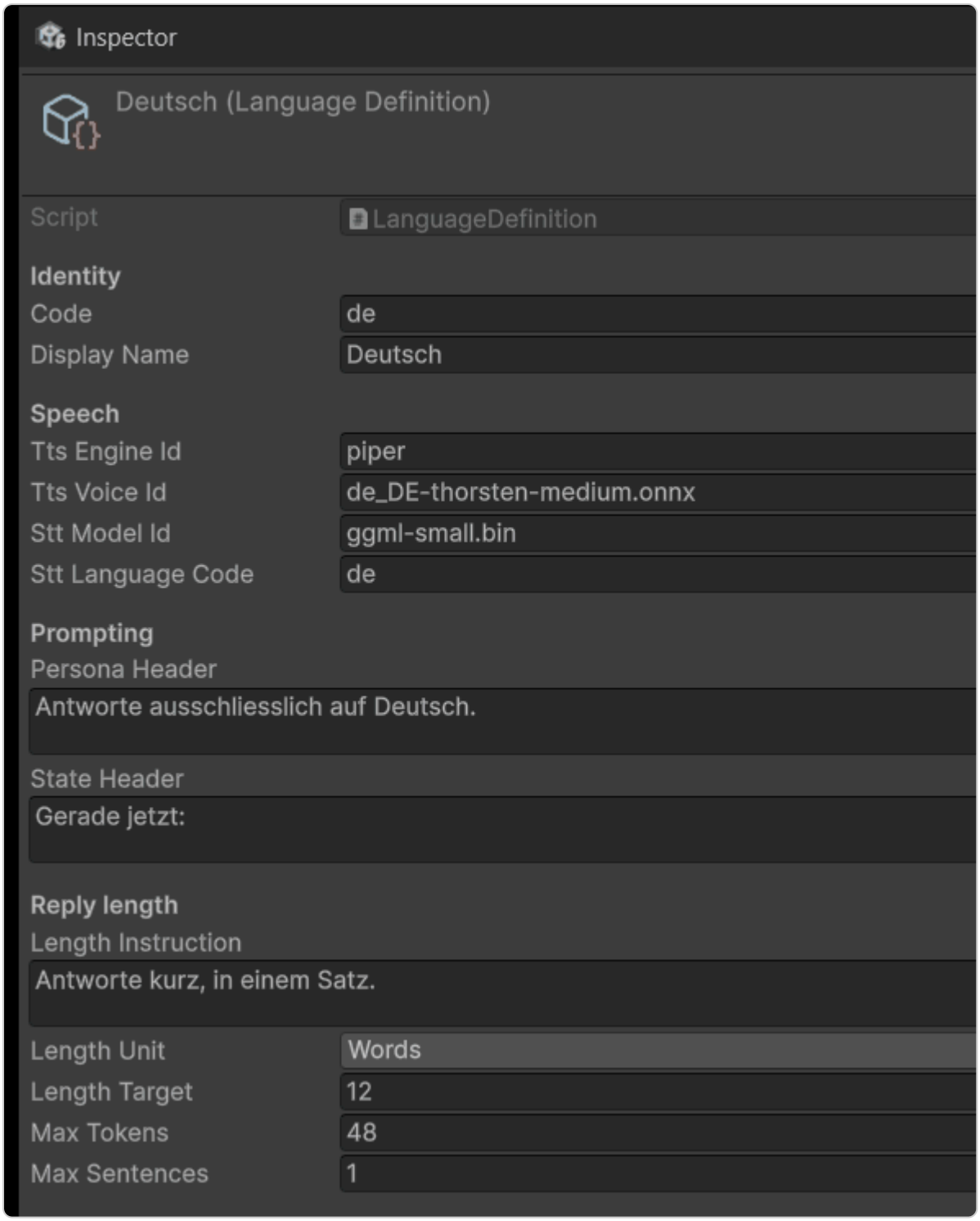
1. **Get a voice.** Piper publishes voices for around fifty languages at `huggingface.co/rhasspy/piper-voices`. **Download both files** — `name.onnx` and `name.onnx.json`. A missing `.json` produces silence with no error.
2. **Put them** in your Piper `voices/` folder.
3. **Create the language asset**: `Create ▸ Offline AI NPC ▸ Language Definition`. Fill in:

Every field on the asset, in the order the inspector shows them. Nothing here is hidden and nothing is optional-but-secretly-required:

Field	Example	Why
Identity		
Code	`ko`	The identity. Keep it stable — everything keys on it
Display name	`한국어`	Shown in your language picker
Speech		
TTS engine id	`piper`	Which engine speaks this language. `piper` unless you added one
TTS voice id	`ko_KR-example-medium.onnx`	The file from step 1. For VOICEVOX, a speaker number
STT model id	`ggml-small.bin`	The recognition model, when it differs per language. Blank = the project default
STT language code	`ko`	Handed to recognition. Usually `Code` without the region part
Prompting		
Persona header	`한국어로만 대답하세요.`	How the persona block is introduced — *in that language*
State header	`지금:`	How world state (time, weather) is introduced. Also in that language
Reply length		
Length instruction	`한 문장으로 짧게 대답하세요.`	The rule the model reads. Leave empty and one is generated (English or Japanese only)
Length unit	`Characters`	**Words are meaningless without word spacing**
Length target	`30`	Characters for CJK, ~12 words otherwise
Max tokens	`48`	The hard stop. This is what actually bounds the reply — the instruction is a request, this is a limit
Max sentences	`1`	Where the reply is cut if the model runs past its instruction anyway

The last two are in bold because Limitations and Troubleshooting both name them as the only thing that genuinely controls reply length. A prompt asking for brevity is not a limit; these two are.

A filled-in asset — German, a language this plugin ships no voice for, which is exactly the case the asset exists to cover:



The voice in that screenshot, `de_DE-thorsten-medium`, is an illustration and **not a recommendation**: apply the provenance test in Licensing to any voice before you ship it, including that one. The plugin cannot know what game you are making.

1. **Add it to your Language Set** and assign that set at startup:

```
csharp LanguageRuntime.Catalog = myLanguageSet.ToCatalog();
```

1. **Bind the voice engine** for the code in `TTSEngineManager`'s binding table. Piper serves any number of languages; only Japanese needs a different engine here.

That is the whole process. There is no list of approved languages in the code, and **the plugin does not gate on this page** — if you have a good Korean voice and a model that handles Korean, add it and it works.

Two things that will bite you

Write the instructions in the target language. An instruction the model cannot read is not an instruction. This was measured: with a Russian directive that only said "no English words", Qwen3-4B still reached for another script in 3 replies out of 12 — `corridors`, `thanks`, once a Chinese word for "tonight". Naming the *script* explicitly — "write only in Cyrillic" — took that to 0 out of 8. Say which alphabet, not just which language.

Count characters, not words, for CJK. "Twelve words or fewer" cannot be evaluated in a language without spaces between words. The length unit exists for this.

Tier 3 — not viable

Languages outside what the model or recognition can do: very low-resource languages, most constructed languages, historical languages, and anything with no meaningful presence in training data. The model will produce something; it will not be good, and no configuration fixes that.

If a language matters to your game and you are unsure which tier it is in, test it before committing. Twenty minutes with the verification window is cheaper than finding out after release.

Switching at runtime

```
LanguageRuntime.SetCurrent("ko");
```

This restarts the speech engine with the new voice and invalidates the prompt cache, so the next reply pays the cold cost (~2.1 s on the reference machine). Rapid changes are coalesced, so dragging a settings dropdown does not queue five restarts. **Switch on a menu screen, not mid-conversation.**

To ask what is configured rather than assuming:

```
if (!LanguageRuntime.IsConfigured("ko"))
    // ... offer the download, or fall back
```

`IsConfigured` answers about the actual catalogue. Do not use `Find` / `FindOrDefault` for this question: they fall back to the default language, so an unconfigured code comes back looking supported. That exact conflation once made a verification matrix report Russian as verified after hearing an English voice say an English sentence.

Per-character voices in one language

A language's voice is the default. A character can override it — that is how two English speakers in the same scene sound different. See Dialogue.

Bring your own engine

The package ships Piper because it is offline, permissively licensed and has a voice whose provenance clears. It is not the only engine you can use, and the seam for adding another is deliberately small.

The contract

```
public interface ITTSEngine
{
    bool IsReady { get; }
    Task StartAsync();
    Task StopAsync();
    IEnumerator Synthesize(string text, int speakerId, Action<AudioData> onAudio);
}

// Optional, when one engine serves several languages:
public interface ILanguageAwareEngine { void SetLanguage(string languageCode); }
```

Four members. `Synthesize` is a coroutine rather than a `Task` on purpose — engines poll, and the queue that owns the voice runs on scaled time, so a wall-clock `Task` would keep speaking through a pause.

`AudioData` is raw samples plus a rate; `AudioClipConversion` turns it into an `AudioClip`. Nothing in the interface mentions Unity, so the engine itself stays testable.

The worked example: VOICEVOX

VOICEVOX is the awkward shape on purpose — an **external application**, started outside Unity, answering over HTTP, with its own idea of when it is ready. If the seam survives that, it survives a DLL or a cloud call.

What the implementation does, before the code itself:

1. **`StartAsync`** does not start VOICEVOX; the player does. It polls `/version` until the server answers, then flips `IsReady`. A health-retry loop rather than a single probe, because the server takes several seconds to boot and a one-shot check would declare it absent forever.
2. **`Synthesize`** is two calls: `POST /audio_query` for the prosody object, then `POST /synthesis` for the WAV. It yields between them, so the frame is never blocked.
3. **`speakerId`** maps straight onto VOICEVOX's speaker numbers — which is why the parameter exists on the interface at all. Piper ignores it (one voice per process); VOICEVOX needs it.

4. `SetLanguage` is where a multi-language engine switches; VOICEVOX is Japanese only, so this example does not implement `ILanguageAwareEngine`.

The one thing worth copying verbatim is the readiness handling. An engine that reports ready before it is produces silence with no error, which is the hardest speech failure to diagnose.

The whole engine, copy this file

```
using System;
using System.Collections;
using System.Threading.Tasks;
using OfflineAINPC.Speech;
using UnityEngine;
using UnityEngine.Networking;

namespace YourGame.Speech
{
    /// <summary>
    /// VOICEVOX as an ITTSEngine: an external HTTP server, four members, no plugin changes.
    /// Drop this component in the scene and bind it to "ja" (see Bind a voice, above).
    /// </summary>
    public class VoicevoxEngine : MonoBehaviour, ITTSEngine
    {
        [SerializeField] private string _baseUrl = "http://localhost:50021";
        [SerializeField] private float _healthCheckTimeout = 2f;
        [SerializeField] private float _healthRetryInterval = 3f;  // re-check cadence while Active +
not yet ready

        public bool IsReady { get; private set; }

        private Coroutine _healthRetry;

        public async Task StartAsync()
        {
            using var req = UnityWebRequest.Get($"{_baseUrl}/version");
            req.timeout = (int)_healthCheckTimeout;
            var op = req.SendWebRequest();
            while (!op.isDone) await Task.Yield();
            IsReady = req.result == UnityWebRequest.Result.Success;
            if (IsReady)
            {
                string version = req.downloadHandler.text.Replace("\"", "");
                Debug.Log($"[VoicevoxEngine] Ready (version: {version})");
            }
            else
            {
                Debug.LogWarning($"[VoicevoxEngine] VOICEVOX server not reachable at {_baseUrl}:
{req.error} — retrying");
                StartHealthRetry();   // server may still be opening port 50021
            }
```

```
        }

        public Task StopAsync()
        {
            // VOICEVOX is user-managed (external process). We don't kill it on language switch.
            IsReady = false;
            if (_healthRetry != null) { StopCoroutine(_healthRetry); _healthRetry = null; }
            return Task.CompletedTask;
        }

        private void StartHealthRetry()
        {
            if (_healthRetry != null) return;
            _healthRetry = StartCoroutine(HealthRetryLoop());
        }

        // VOICEVOX.exe may still be opening port 50021 when we boot in JP. Instead of
        // muting Japanese TTS for the whole session, re-poll /version until it answers
        // (or until StopAsync stops us on a switch away). Mirrors the Piper watchdog:
        // IsReady ends up meaning the same thing in both ITTSEngine implementations.
        private IEnumerator HealthRetryLoop()
        {
            while (!IsReady)
            {
                yield return new WaitForSeconds(_healthRetryInterval);
                using var req = UnityWebRequest.Get($"{_baseUrl}/version");
                req.timeout = (int)_healthCheckTimeout;
                yield return req.SendWebRequest();
                if (req.result == UnityWebRequest.Result.Success)
                {
                    IsReady = true;
                    Debug.Log($"[VoicevoxEngine] Ready after retry (version:
{req.downloadHandler.text.Replace("\"", "")})");
                }
            }
            _healthRetry = null;
        }

        public IEnumerator Synthesize(string text, int speakerId, Action<AudioData> onAudio)
        {
            string queryUrl = $"{_baseUrl}/audio_query?text={UnityWebRequest.EscapeURL(text)}&speaker=
{speakerId}";
            using var query = UnityWebRequest.PostWwwForm(queryUrl, "");
            yield return query.SendWebRequest();
```

```
            if (query.result != UnityWebRequest.Result.Success)
            {
                Debug.LogError($"[VoicevoxEngine] query failed: {query.error}");
                yield break;
            }

            string synthUrl = $"{_baseUrl}/synthesis?speaker={speakerId}";
            byte[] body = System.Text.Encoding.UTF8.GetBytes(query.downloadHandler.text);
            using var synth = new UnityWebRequest(synthUrl, "POST");
            synth.uploadHandler = new UploadHandlerRaw(body);
            synth.downloadHandler = new DownloadHandlerAudioClip(synthUrl, AudioType.WAV);
            synth.SetRequestHeader("Content-Type", "application/json");
            yield return synth.SendWebRequest();

            if (synth.result != UnityWebRequest.Result.Success)
            {
                Debug.LogError($"[VoicevoxEngine] synth failed: {synth.error}");
                yield break;
            }

            AudioClip clip = DownloadHandlerAudioClip.GetContent(synth);
            var audio = AudioClipConversion.FromClip(clip);
            if (clip != null) UnityEngine.Object.Destroy(clip);
            onAudio?.Invoke(audio);
        }
    }
}
```

`AudioClipConversion` and `AudioData` come from `OfflineAINPC.Speech`; nothing else here is plugin-specific. Bind it to Japanese and the rest of the pipeline — queue, ducking, lip sync, subtitles — does not know the difference.

Licensing: apply the test, do not take a list

This page will not tell you which VOICEVOX voices are safe for your game, and you should distrust any page that does. Terms are per character, live on the rights holders' own pages, change, and several restrict the *kind of content* rather than only commercial use. The plugin cannot know what game you are making.

What travels between projects is the test, and by now you have seen two shapes of it:

Engine	Where the answer lives	What caught people out
Piper	the **Training line** of the model card	`libritts` and `libritts_r` are one letter apart, same corpus, opposite conclusion
VOICEVOX	the **character's own terms page**	attribution is mandatory and names the character; some characters require an application for commercial or corporate use; some restrict content

Both are the same question asked twice: *what is this made of, and who could object?* See Licensing for the long form.

Hardware tiers

The plugin probes the machine at startup, picks a tier, chooses a model for it, sizes the GPU layer count against the real VRAM budget, and steps down if a load fails. Your game does not have to ask any of that.

What it decides is printed once at startup:

```
Hardware: NVIDIA GeForce RTX 3060 Laptop GPU (5996 MB VRAM, discrete), 32498 MB RAM, 16 threads
Tier resolved: Standard — 3996 MB VRAM free after a 2000 MB game reserve
Startup verdict: tier Standard; model qwen3-4b-q4km, 36 GPU layers
```

The tiers

Tier	Machine	Model size	What the player gets
Premium	≥ 10 GB VRAM	7–8B	Best writing, full context
Standard	≥ 3.5 GB VRAM	3–4B	The reference configuration below
Light	below that, or ≥ 16 GB RAM and ≥ 4 cores with no usable GPU	1–2B	Works, noticeably more stilted, slower
Fallback	anything less	none	**No language model.** A working state, not a crash

Thresholds are VRAM *after* a reserve for your own rendering — 2000 MB by default, plus more in VR. All of it is editable in a `TierPolicy` asset; the defaults are a starting point, not a law.

The reference machine, which produced every latency figure in this documentation: RTX 3060 Laptop, 6 GB VRAM, 32 GB RAM, 16 threads → **Standard**, Qwen3-4B Q4_K_M, all 36 layers on the GPU, 36 tok/s, **2.0 s median to first audio**.

Fallback is a design decision, not an error

On a machine that cannot run any model the plugin resolves to Fallback and **your game must still work**. Nothing throws; the character simply has no generated dialogue.

> Characters will not generate speech on this machine. Fall back to your own written lines.

Decide early what that looks like in your game — authored lines, a text-only mode, or a message. A game that assumes the model is always there will ship broken for the low end of its own audience.

```
if (Tiers.Current == HardwareTier.Fallback)
    UseAuthoredDialogue();
```

`Tiers.Current` is `HardwareTier?`, and it is null until a tier has actually been resolved — which does not happen until the first time the game runs. The comparison above is safe (a null never equals `Fallback`), but anything that needs the value must say so:

```
if (Tiers.Current is HardwareTier tier)
    Debug.Log($"Running at {tier}");
else
    Debug.Log("The machine has not been measured yet.");
```

It is nullable on purpose. "Nobody has measured this yet" and "this machine cannot host a model" are different facts, and a property that answered `Fallback` to both would send code written exactly like the snippet above to authored lines on hardware that runs the model perfectly well. Null means the question is still open; ask again after the cascade has run.

Forcing a tier while developing

You cannot test the low end on a machine that never resolves there. Set the PlayerPrefs key `OfflineAINPC.ForceTier` to a tier value (`-1` disables), and `OfflineAINPC.FailStarts` to make the next N start attempts fail so the downgrade cascade is exercised on purpose.

Test Fallback before you ship. It is the tier most likely to reach a player and least likely to have been run.

What "degrades" actually means

A lower tier is a smaller model. It is not merely slower — it writes worse: shorter, more literal, more repetitive, and worse at holding a persona across a long conversation.

Measured on the Light tier, 2026-07-30 (Qwen3-1.7B-Q8_0, 28 GPU layers, 20 sessions of 16 turns = 320 turns, same brief and questions as the 4B run):

	Light (1.7B)	**Standard (4B)**
Departures from stated facts	**0 of 320**	0 of 320
Median time per reply	260 ms	301 ms
p90	781 ms	614 ms
Median reply length	45 chars	—

These milliseconds are not the 2.0 s quoted above, and the two are not comparable. This harness measures generation alone — text out of the model, no speech synthesis, no audio device. The 2.0 s is the whole chain a player experiences: prompt, generation, synthesis, and the first sample reaching the speakers. A reply generated in 301 ms still takes about two seconds to be *heard*.

The small model holds a brief exactly as reliably as the larger one. What it loses is manner, not memory: it answers in flat, complete sentences ("I have worked at Halden Station for three years") where the 4B is terser and more in character.

So a design that asks the model to REMEMBER survives this tier, and a design that asks it to PERFORM does not. That is the practical meaning of the advice above.

A design that leans on the deterministic half (state, thresholds, events) degrades gracefully, because that half is identical on every tier. A design that leans entirely on the model's writing quality degrades badly. That is worth knowing before you build the game around it, not after.

Recipes

Five shapes this plugin is actually good at, with what each one uses and roughly what it costs to build. Times assume you have done the quickstart once and are comfortable in Unity; they are for the wiring, not for your art or your writing.

---

1. Replacing a dialogue tree, in an afternoon

What it is. One character who answers anything, instead of three canned options. The fastest route from import to something worth showing.

Uses. Dialogue + memory. Nothing else.

Configure. A persona (who she is, how she speaks, three or four facts), a voice, a push-to-talk key or a text field.

About two hours, most of it writing the persona.

Watch for. Worded length limits do not work on small models — set the token ceiling. And build the system prompt once, not per turn, or every reply pays the 2.1 s cold cost.

---

2. A companion who reacts to the room

What it is. She notices what is actually around her, and mentions it unprompted.

Uses. Dialogue + **perception**.

Configure. A `PerceptionQuery` on the character, POI components on the objects worth noticing, a description per object, occluder layers set to your walls.

Half a day, mostly tagging objects.

Why it matters, measured. Same scene, same model. *Perception off:* she mentioned a fireplace, a window seat and a cat — none existed. *Perception on:* she spoke only about real objects. This is the single highest-value system in the package for making a character feel present.

Watch for. Pivots. An object whose pivot sits on the floor while its body is a metre up traces occlusion from the floor and reads as hidden. This made half of one scene's furniture invisible.

---

3. An interrogation scene

What it is. The player asks anything; the character has something to hide; how open she is changes as the conversation goes.

Uses. Dialogue + **state** (one axis) + threshold events.

Configure. One axis with three bands and the game events that move it. Start from the short path — ten lines, no assets.

A day, including the visible consequences.

The rule that makes it work. Deterministic causes, deterministic consequences. The player presses, the axis drops, and *you* change posture or lighting on the band change. State colours her speech about six times in ten; the posture is exact every time. Build the loop on the exact half.

Also: put the checkable facts in the character's brief and the discrepancies in the player's evidence — not in the model's reliability. Measured over 320 turns, a 4B held its brief with zero unforced departures; it is a reliable witness to what you told it, and an unreliable arithmetician (a derived figure came out wrong in 19 sessions of 20, the same way every time). Never make the player check the model's arithmetic.

---

4. A shopkeeper or quest-giver

What it is. She does things, not only says them — points at stock, opens the ledger, marks a quest.

Uses. Dialogue + **actions**.

Configure. Register a few verbs with their targets; wire each to a method. Grammar makes the payload well-formed, so you parse a struct, not prose.

Half a day.

Watch for. The verbs go in the cached system prompt; the *targets* arrive per turn, because what she can reach changes as she moves. Baking the targets into the system prompt means she names objects that are no longer there.

---

5. A roommate or life-sim character

What it is. Someone with moods and needs that move over hours, who is different at midnight than at noon.

Uses. Dialogue + **state** (several axes, traits, persistence).

Configure. An axis catalogue, a trait preset per character, a persistent store, and a coarse tick.

A day or two, most of it tuning rates so a day of game time feels right.

Watch for. Two characters with the same catalogue and no trait presets behave identically, which reads as one character with two faces. Traits are what separate them.

And: persistence needs a store you supply. The plugin ships none, deliberately — your save system is yours.

What none of these are

A crowd. Every recipe here has **one character generating at a time**. Two at once costs about 5× on the player's own reply. See Is this for your game?.

Limitations

Every limit here is stated with its mitigation where one exists, and stated bare where none does. The goal is that you are never surprised — not that every sentence carries a disclaimer.

All figures come from measurements on an **RTX 3060 Laptop (6 GB VRAM), 32 GB RAM, Qwen3-4B Q4_K_M with all 36 layers on the GPU**. Your numbers will differ; the tooling that produced these is in the package (`Tools ▸ Offline AI NPC ▸ Live verification`) so you can produce your own.

Concurrency

The limit. All characters share one model server and requests serialise. Measured, same mode both rows:

	Median to first audio
One character generating	2.4 s
Player's turn while background characters generate	**12.7 s**

Roughly 5×. The worst single turn in that contended run took 125 s.

The mitigation. Suspend background conversation the moment the player starts speaking, rather than when their speech has been transcribed — the transcription itself takes seconds you have already lost.

```
// The player pressed push-to-talk. Everything else stops generating NOW.
_pushToTalk.RecordingStarted += () => _ambientChatter.Suspend();
_dialogue.ReplyFinished     += () => _ambientChatter.Resume();
```

Suspend on the *input* event, not on the transcript — that single change is the difference between the two rows above. Waiting for the transcript costs you the whole recognition pass before the characters stop talking.

What has no mitigation. Two characters genuinely talking to each other while the player also talks. One of the three waits. Design around it.

Latency

Measured over 21 warm turns, one character, nothing contending, through the shipping path (speech starts at the first finished sentence):

Median to first audio	**2019 ms**
p10–p90	1545–2761 ms
Range	1232–3013 ms
First turn of a session (cold)	**2.1 s**

Measured 2026-08-16, 21 warm turns, English, the streamed path, on a **cleared history** — the run is kept as `docs/measurements/latency_clean-history_streamed_4B_2026-08-16.txt`, and its header records the history it started from.

The slowest warm turn in twenty-one took 3013 ms, and the spread is narrow — but design for the tail rather than the median: a design that cannot absorb an occasional three-second pause will feel broken on those turns.

Mitigations. - The cold first turn is prompt-cache warm-up. Fire one throwaway generation during a loading screen or a scene fade and the player never meets it. - Give the character something to do while generating — a glance up, a breath, a posture change. Two seconds of a character visibly thinking reads very differently from two seconds of a frozen face. - Keep the system prompt stable. It is cached; changing it mid-session pays the cold cost again. See Dialogue.

What has no mitigation. The floor is the floor. If you need sub-second responses, this is the wrong technology, not the wrong configuration.

Speech recognition is a second wait, and it is the larger one

Every latency figure above is measured **from the moment text is submitted**. If your player speaks rather than types — the configuration this plugin exists for — transcription happens first, and it is not free.

Whisper-small, one short spoken phrase	**7908 ms and 8004 ms**
Measured	in the interview demo, 2026-08-16, while the language model held the same GPU
What the player experiences	that, and then the figures above

Two things follow, and neither is a defect in the plugin:

- **The model size is your decision, and it is a real trade.** `tiny` and `base` transcribe several times faster and get more words wrong; `small` is what these numbers came from. Wrong words are worse than they sound here, because the transcript is what the character answers — a misheard question produces a confident answer to something nobody asked.

- **It shares the GPU with the language model.** These numbers were taken with both resident, which is the shipping configuration; recognition alone on an idle machine is faster.

The plugin reports the generation half through `StreamingSpeechSession.TimeToFirstAudioMs`, measured from the first unit enqueued. Recognition happens before that and belongs to your code, so time it there if you want to show the whole trip.

Switching language mid-session

Changing language stops and restarts the speech engine with a different voice model, and invalidates the prompt cache. Rapid switching is coalesced — only the final selection is applied — so a settings dropdown being dragged does not queue up five restarts.

How much the prompt-cache half costs was measured, and the honest answer is "it depends on your prompt, and often it is lost in the noise". Six passes, 4B, short system prompts: the first turn after a switch came out anywhere from 1080 ms FASTER to 2550 ms slower than the warm average, because reply-length variance is larger than the effect. With this project's full system prompt — persona, memory replay, action verbs — the first turn of a session costs about 2.1 s, and a language switch pays that again.

The rule to take away is not a number: **the bigger your system prompt, the more a language switch costs, so switch on a menu screen rather than mid-conversation.**

Desktop only

Windows, macOS and Linux. Not mobile, not web, not console. The model file is gigabytes and the speech engines are external processes; neither survives those platforms.

No mitigation. It is not on a roadmap.

Output is probabilistic

A language model produces different text every run, occasionally something off-key, and it will not deliver an authored line verbatim.

Mitigations. - Grammar constrains the *shape* of a reply — one sentence, a well-formed action payload — reliably, because tokens that would break the shape have no probability left. See Actions. - The token ceiling and sentence limit are what actually keep replies short. **Small models ignore worded length instructions**: "reply in one short sentence" in a persona is a suggestion, the ceiling is not. Tune the ceiling, not the wording. - For lines that must be exact, author them and use the model for the space between.

What has no mitigation. You cannot guarantee any particular sentence. Plan for it.

State affects speech only sometimes

This one is easy to overstate in either direction, so precisely:

	Reliability
A state value, its band, and its threshold event	**Exact, every time**
A state descriptor changing how the model *speaks*	**~6 times in 10** (0 with the system off)

Measured by asking the character directly about her state. The deterministic half is a mechanism you can build on; the speech colouring is a bonus that does not always arrive.

The rule that follows: every visible consequence of state must have a deterministic cause. If a character's tiredness matters, show it in posture, pacing or an event — and let the speech colouring be the extra. A design whose core loop rides on the probabilistic half will feel unreliable exactly when someone is judging it.

Perception is opt-in, and matters more than it sounds

Without it, characters invent surroundings. Measured, same scene, same model:

- **Perception off:** the character mentioned a fireplace, a window seat and a cat. None existed.
- **Perception on:** she spoke only about objects that were actually in the room.

Mitigation: turn it on. See Perception. The cost is a coarse-timer recompute, not per-frame work.

Voices come in pairs, and a missing partner is silent

Every Piper voice is two files: `name.onnx` and `name.onnx.json`. **If the `.json` is missing, synthesis produces silence and no error** — the engine starts, reports ready, and says nothing.

Mitigation: the setup checker (`Tools ▸ Offline AI NPC ▸ Check my setup`) reports an unpaired voice as a problem. Run it after adding any voice.

Pause: what each part does at `timeScale = 0`

Decided per subsystem, because the right answer differs.

Subsystem	Clock	At `timeScale = 0`
State axes (decay, needs)	scaled	**Freeze.** She does not get sleepier while the player is in a menu
Perception	scaled	**Freezes**, consistently with state
Speech queue	**unscaled**	**Keeps running** — see below
Speech-process watchdog	unscaled	**Keeps running.** An external process lives on wall clock
Generation already in flight	off the game loop	**Continues**, deliberately

The speech queue is the non-obvious one. `AudioSource.Play` does not obey `timeScale`: a line already sounding keeps sounding through a pause. Every timer around playback is therefore unscaled, so the bookkeeping stays beside the sound that is actually playing.

Scale those timers instead and the failure is quiet: audio runs while the code waiting for it stands still, the subtitle hangs on screen, the queue never advances and the session never ends. Worth knowing if you write your own queue.

Generation is not cancelled on pause because it has already been paid for, and cancelling would cost a cold prompt cache (~2.1 s) on resume for no benefit. If your game wants a pause to abandon the reply, call `Cancel()` on the session yourself.

Behavioural verification of this needs Play mode. The decisions above are implemented and reasoned; watching a menu open mid-reply is not something a unit test can do.

Verification needs Play mode for dialogue

Voices can be auditioned from the editor without entering Play mode. **Dialogue cannot.** The language model is constructed in `Start()` and its continuations only run inside the player loop, so a live dialogue check requires Play mode.

No mitigation — it is how Unity's lifecycle works. Documented so the tooling's scope is not mistaken for a bug.

Not included

Named so you do not go looking:

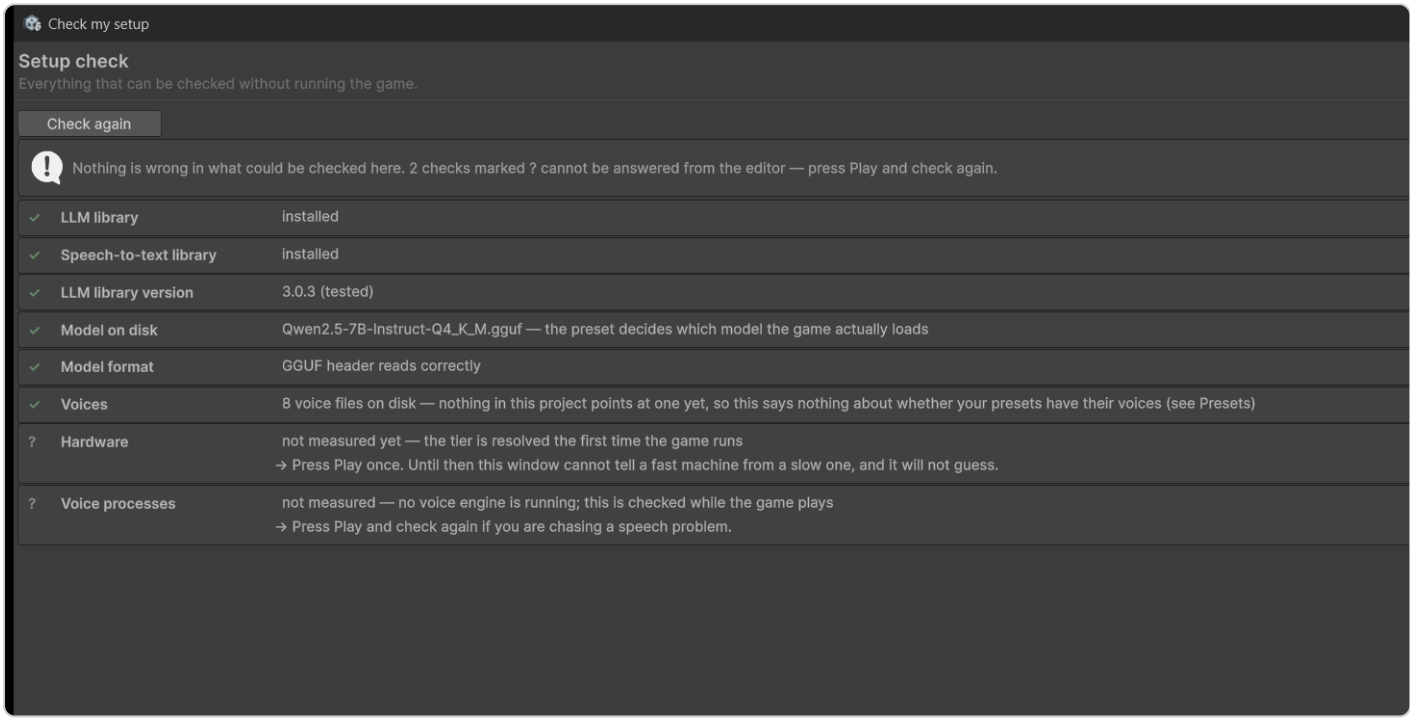
- **RAG / document retrieval.** Not present.
- **Lip sync.** The demo project drives a jaw bone itself; the plugin does not ship a solution.
- **Emotion or voice-style control in TTS.** Piper voices have a fixed delivery.

- **Streaming speech recognition.** Recognition runs on a finished utterance, not continuously.
- **Turn history in the prompt debugger.** The debugger exists and IS broken down by section — a static-prefix column and a this-turn column, each section collapsible with its own token count. What it does not show is previous turns: you see the prompt for one turn, not the conversation that led to it.

Troubleshooting

Ordered by how often it happens. Every message below is a string the plugin actually emits — search for it here before searching the internet.

First, always: `Tools ▸ Offline AI NPC ▸ Check my setup`. It reports what is missing and what to do about it, and it distinguishes "this is broken" from "this could not be determined" rather than guessing.



This is what a failure looks like: the check that failed is red, and the line under it is the fix rather than a restatement of the problem. Match your window to this one before reading further — most of what follows is the long form of a line in that list.

Four markers, and the fourth is the one worth knowing:

	Means
✓ green	checked, and fine
! amber	works, below the intended quality or speed
✕ red	cannot work until you fix it
? grey	**nobody has measured this yet** — not a pass

The grey ones are questions the editor cannot answer: your hardware tier before the game has ever run, the speech processes before one is up. Press Play once and check again. Grey is deliberately not green — "not measured" and "all good" must not look identical.

The character speaks in the subtitle but there is no sound

By far the most common, and it produces no error at all.

> A Piper voice is a pair of files. Download both; the model alone loads and then says nothing.

A Piper voice is `name.onnx` **and** `name.onnx.json`. With the `.json` missing the engine starts, reports itself ready, and synthesises silence.

Fix: download both files into the `voices/` folder. The setup check flags an unpaired voice.

No model weights found

> `[OfflineAINPC] No model weights found. Expected one of: …`

The tier system resolved a tier, asked the catalogue for that tier's model, and the file is not on disk.

Fix: open `Tools ▸ Offline AI NPC ▸ Model Manager` and download the preset for your language and tier, or place the `.gguf` in `StreamingAssets` under the exact file name the message lists. A renamed file is not found — the name in the manifest is the name on disk.

The character never says anything, and push-to-talk stops working

Almost always the busy lock. If your controller sets a `_busy` flag and an early return, an exception or a cancelled coroutine skips clearing it, the character is locked out for the rest of the session, silently.

Fix: release it in a `finally`, on every path. See Dialogue. This is the single most common integration bug in this codebase's own history.

Replies are much slower than the documented figures

Check, in this order:

1. **Is something else generating?** Background conversation costs about 5× — see Limitations.

2. **Is this the first turn?** The first reply of a session is ~2.1 s while the prompt cache fills. Warm it during a loading screen.
3. **Is the prompt changing every turn?** A changed system prompt discards the cache and pays the cold cost again. Build it once.
4. **Which tier resolved?** The startup log says so: `Startup verdict: tier Standard; model qwen3-4b-q4km, 36 GPU layers`. A machine that fell to a lower tier is running a smaller model on fewer GPU layers, and will be slower per token than the reference figures.

The engine is not ready, or the process died

> A speech process died or failed to start. The plugin restarts one automatically within its restart budget.

Piper occasionally dies mid-synthesis. Recovery is automatic: a dead voice is re-spawned on its next request, and a watchdog re-spawns the default voice. The budget is three restarts per minute; past that the engine reports itself not ready and logs once rather than looping.

What you lose: the reply in flight, not the session.

If it happens constantly: the voice file is likely corrupt or truncated. Re-download it and check the file size against the model card.

VOICEVOX says it is not ready

VOICEVOX is a separate application with an HTTP server, and it takes time to boot. The plugin retries a health check and flips to ready when the server answers.

Fix: start VOICEVOX before entering Play mode, or wait — it will connect on its own.

A dependency is missing

> Install LLM for Unity (package id `ai.undream.llm`). Without it there is no model to talk to.
>
> Install whisper.unity (`com.whisper.unity`) to accept voice input. Typed input works without it.

Both are external and free. The plugin compiles without either — the features that need them are compiled out through `versionDefines`, so a missing dependency is a missing feature and never a compile error.

> It may work. If generation misbehaves, try one of the tested versions before reporting a bug.

You are on a version this plugin has not been tested against. Usually fine.

The character talks about furniture that is not there

Perception is off. Measured in the same scene: with it off the character mentioned a fireplace, a window seat and a cat, none of which existed; with it on she spoke only about real objects.

Fix: see Perception. Also check pivots — an object whose pivot sits at `y = 0` while its body is a metre up traces occlusion from the floor and reads as hidden. That made half of one scene's furniture invisible to perception.

The character ignores "reply in one short sentence"

Expected. **Small models do not honour worded length limits.** The token ceiling and the sentence limit are what actually keep replies short.

Fix: lower the ceiling in the language definition's length rule, not the wording in the persona.

A language switch produces the wrong voice, or an English reply

Two different causes:

- **Wrong voice, right language:** the language has no voice bound for its code in the engine binding table. The manager logs which codes it has: `No engine bound for 'ko'. Bound languages: ja, en, es, ru`.
- **Right voice, wrong language of reply:** the voice was switched but the prompt was not. Switch through your language manager so the system prompt is rebuilt, not through the TTS manager alone. This one hides well — the voice is correct both times, so the switch looks like it worked and only the words are wrong.

The transcript comes back in a script nobody spoke, or as bracketed sound words

> the player said "do you hear me" and the box filled with `(シャッ)(シャッ)(シャッ)(シャッ)`

Two separate faults, and they arrive together often enough to look like one.

- **Wrong script:** `WhisperManager.language` is set to a language the player is not speaking. Recognition does not refuse a mismatch — it transcribes what it heard using the language it was told to expect, so English audio under a Japanese setting comes back as Japanese sound-words. Set the language to what your

players speak; if you copied your speech setup from another scene, check this field first, because everything else in that setup copies correctly and only this one is scene-specific.

- **Bracketed sound words:** recognition annotates non-speech — `(coughs)`, `[BLANK_AUDIO]`, `*sighs*`, `♪` — and a run of those is not short, not silent, and not a known hallucination, so `UtteranceGate` passes it. **The gate does not strip annotations, by design:** it judges whether somebody spoke, and someone who coughs before a real question did speak. Deciding what to do with the annotation is the caller's, because a game that shows a transcript and a game that feeds one to a model want opposite things.

Fix for the second: strip annotations before you call `Inspect(text, language)`, and treat a line that is *only* annotation as nothing said. Removing the brackets rather than rejecting the line matters — `(coughs) where were you on the fourteenth` is a real question with a cough in front of it. The demo does this in `Transcript.Clean`, which is thirty lines of pure C# you can copy.

Verification says a language is not configured

> `'ko' is not one of the languages this project configures (it configures en, ja, es, ru)`

Correct and deliberate. Ask `LanguageRuntime.IsConfigured(code)`, which answers about the catalogue. Do not use `Find` / `FindOrDefault` for that question — they fall back to the default language, so an unconfigured code comes back looking supported.

Live verification cannot check dialogue

Voices audition from the editor. Dialogue does not: the language model is constructed in `Start()` and its continuations only run inside the player loop.

Not a bug. Enter Play mode for dialogue checks.

Still stuck

`Tools ▸ Offline AI NPC ▸ Diagnostics` produces a report you can paste into a support request: hardware, tier, model, dependency versions, what is installed and what is missing. It contains no personal data and is not sent anywhere — the plugin has no telemetry of any kind.

Licensing

What you may ship, and what shipping it obliges. Read this before release, not after.

None of it applies to the plugin itself — `Assets/OfflineAINPC/` is proprietary and yours to use under the Asset Store licence. Everything below is about the things the plugin *talks to*, which you install and may choose to redistribute.

The one that can bite: espeak-ng

Piper's phonemiser links **espeak-ng, which is GPL-3.0**. Piper's own MIT licence does not change that: a binary distribution of `piper.exe` carries espeak-ng's terms with it.

The plugin is not affected. It starts Piper as a separate process and talks to it over a pipe; it does not link, load or contain any espeak-ng or Piper code, and nothing under `Assets/OfflineAINPC/` is a copy of either. A build check fails if that ever changes.

Your game may be affected, if you ship Piper inside your build. That is allowed and normal. It obliges three things:

1. **Ship the licence text**, naming espeak-ng as the component it covers.
2. **Offer the corresponding source** — a URL to the exact upstream release you shipped, kept available for the period the licence requires. Pin a release tag; "latest master" is not corresponding source.
3. **Add no further restrictions.** Your EULA must not forbid, for that component, what the GPL permits.

Your own game stays proprietary. Launching a process does not make your code a derivative work any more than calling a compiler does.

The escape hatch: do not ship Piper. Point the plugin at VOICEVOX, at a system voice, or at your own engine, and let players who want Piper install it themselves. Nothing in the plugin requires it.

Voice models — provenance, not just the licence line

A voice's dataset licence is not the whole answer. Read its Training line too.

Most Piper voices are fine-tuned from the `lessac` voice, and the lessac dataset licence (Blizzard 2013, CSTR Edinburgh) excludes commercial speech-synthesis products from its permitted "Research Purposes". A permissive dataset does not clear that: the licence covers the recordings the voice was fine-tuned ON, not the voice it was fine-tuned FROM.

The distinction, in one pair of examples

These two are the same corpus under the same licence, and only one is usable:

Voice	Dataset	Licence	Training line	Verdict
`en_US-libritts-high`	LibriTTS train-clean-360	CC BY 4.0	**"Trained from scratch on train-clean-360"**	usable
`en_US-libritts_r-medium`	LibriTTS-R	CC BY 4.0	**"Fine-tuned from English lessac medium"**	tainted

Identical dataset name, identical licence, opposite answers. **The Training line decides.** Apply that test to any voice you are evaluating.

What this package ships with

Voice	Licence	Provenance	Where
`en_US-ljspeech-medium`	**public domain** (LJ Speech)	"Trained from scratch for 1000 epochs"	en standard, en light
`en_US-ljspeech-high`	**public domain**	same, higher quality	en premium

The default line-up owes no attribution at all, which is deliberate — a requirement nobody notices is how a game ships in breach of a licence it meant to honour.

If you want many distinct voices

`en_US-libritts-high` carries **904 speakers** selected by id, trained from scratch on LibriTTS, **CC BY 4.0**. It is not in the default presets because CC BY requires attribution and the defaults are deliberately obligation-free. Use it deliberately, and display:

> Speech synthesis by Piper (MIT). Multi-speaker English voice "libritts" trained from scratch on LibriTTS train-clean-360 by Heiga Zen et al., licensed CC BY 4.0 (https://creativecommons.org/licenses/by/4.0/).

Audition speakers with `Tools ▸ Offline AI NPC ▸ Speak one line ▸ Audition a LibriTTS speaker...`.

Every other language

The slots are empty, and the reason is precise: **no voice with a traceable commercial provenance was found** — the available Piper voices for those languages are fine-tuned from the research-only base. That is a supply problem, not a plugin limitation. Supply your own voice or commission one; see Languages.

Dated copies of every card read, with the full reasoning: `LICENSES/VOICE_LICENCES.md`.

VOICEVOX character voices carry per-character terms that bind the **end user**, not you. Surface those terms rather than absorb them — a link in your credits, not a line in your EULA.

Does espeak-ng's GPL reach the audio it produces?

No, and this is worth stating plainly because buyers ask.

The FSF's position is that a program's **output is not covered by the copyright on the program's code**. Synthesised speech is output; it is not a derivative work of espeak-ng, and you owe nothing on the audio your game plays.

What the GPL does reach is **distribution of the binary**. That is the boundary this plugin is built around, and running Piper as a separate process — which it already does — is the recognised way to keep copyleft away from your own code. The obligations in the section above apply to shipping `espeak-ng.dll`, never to the wav files it helped make.

Model weights

The default catalogue is **Apache-2.0** (the Qwen3 family), chosen because it is the cleanest licence to redistribute under and needs no visible attribution.

Some other families do require attribution — Llama's "Built with Llama" is the usual one. The manifest has a field for it and the inspector surfaces it as a finding, because a required attribution nobody notices is how a game ships in breach of a licence it fully intended to honour. **A line in your documentation does not satisfy it; it has to be visible in the product.**

Weights are never bundled with this package. It describes a download; it is not one.

What the plugin itself contains

No DRM, no licence check, no phone-home, no registration, no time limit, no watermark, and **no telemetry of any kind — not even anonymous**. "No data ever leaves the machine" is the product's core promise, and a usage counter would make it a lie.

One practical consequence, stated plainly: nothing reports back, so nobody can tell how the plugin is being used. That is the intended trade.

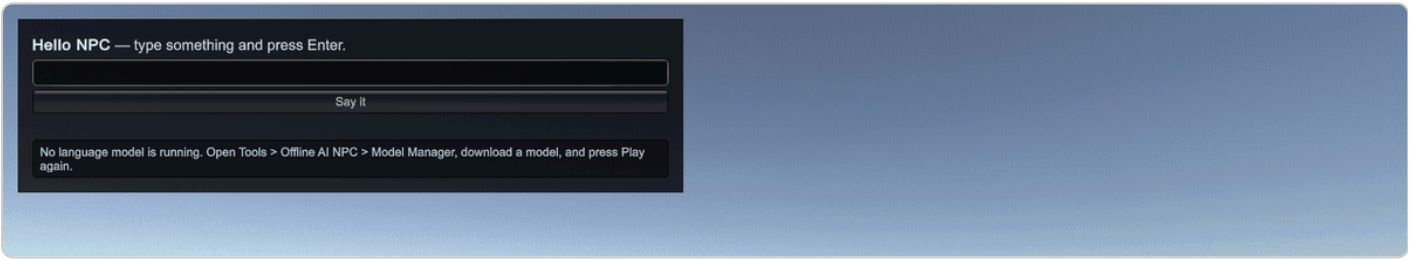
Full third-party inventory: `LICENSES/THIRD_PARTY_NOTICES.md`.

Hello NPC

One character, one text box, one voice. This is the smallest complete thing the plugin does, and it is the sample to read first.

Run it

1. Install **LLM for Unity** (`ai.undream.llm`) if you have not. This sample is the one part of the package that needs it to compile at all — it talks to `LLMAgent` directly, which is the point of it. Without the package the sample's assembly is skipped rather than failing: the rest of the plugin, and the other two samples, compile and run as normal.
2. Open `HelloNpc.unity` .
3. Press Play.
4. Type into the box and press Enter.



If a model or a speech engine is missing, the scene says so on screen and names the window that fixes it. It opens and runs either way — you can read the whole sample without downloading anything.

What to look at

`HelloNpcCharacter.cs` is the whole thing, and it is the file you are meant to copy. It wires four pieces the documentation describes separately:

Piece	Its job
`LLMAgent`	the conversation and the persona
`StreamingReplyParser`	cuts the token stream into speakable units
`StreamingSpeechSession`	synthesises the next unit while the current one plays
your subtitle	driven by the SPEECH, not by the token stream

Three things in that file are not obvious, and each one was a real bug here:

- **The streaming callback is cumulative.** Every call hands you the whole reply so far, not the new piece. Treating it as a delta makes her say `MMaraMara VMara Vey`.
- **Speech is a queue.** She starts talking after the first sentence, not after the last one. Most of the felt latency is this and nothing else.
- **The subtitle follows the audio.** Text is ready seconds before the sound for it; a subtitle wired to the stream runs ahead of her mouth and reads as a bug.

What it needs

- A model — `Tools > Offline AI NPC > Model Manager`.
- A voice — Piper, with **both** `name.onnx` and `name.onnx.json`. A missing `.json` gives silence with no error.

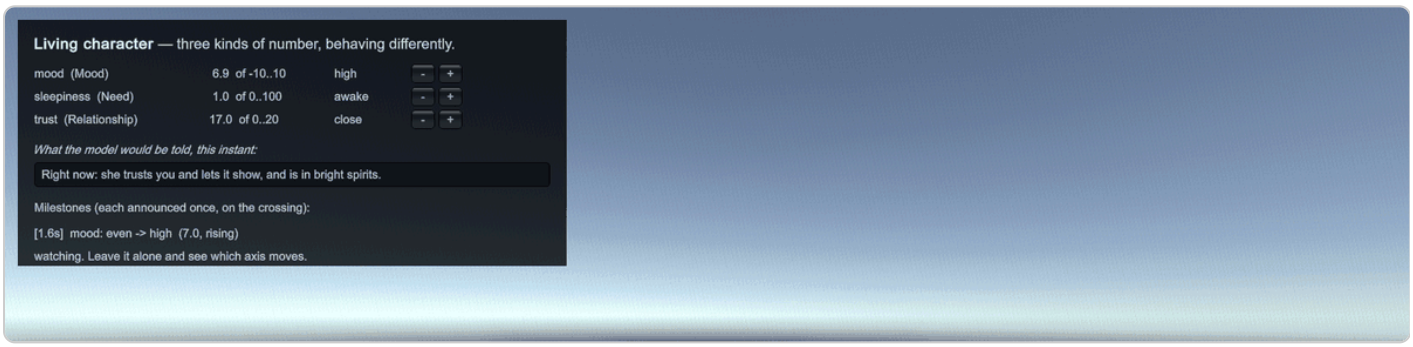
Neither is needed to open the scene.

Living character

Mood, needs and a relationship on one screen, behaving differently — which is the whole argument for keeping them apart.

Run it

Open `LivingCharacter.unity` and press Play. No model needed: this sample is about the state system, and none of it asks anything of a language model.



What to notice, in this order

1. **Leave it alone.** Only `sleepiness` climbs. A need grows until something satisfies it; a mood does not, and a relationship does not drift at all.
2. **Push the mood up and let go.** It eases back to the character's baseline by itself. The speed is her *trait*, not a global setting — swap the trait asset and the same nudge decays at a different rate.
3. **Press `+` on trust until a band changes.** The milestone fires once, on the crossing. Nothing polls, nothing remembers a previous value, and nothing compares against a number a caller had to keep in step with the asset.

The box in the middle is the real thing: the exact sentence the model would be handed this instant. Never a number — a small model given `trust: 7` either ignores it or plays it as a crisis, so the designer writes the words for each band and the model gets those.

Files

- `LivingCharacterAxes.asset` — the axes. Ranges, rates, bands and their words all live here, so tuning is not a code change.
- `LivingCharacterTrait.asset` — one disposition. Shifts the baseline, scales the rate, scales how hard incoming changes land.
- `LivingCharacterDemo.cs` — the screen. Roughly eighty lines, and most of it is layout.

World aware

A character who knows what is in the room and can act on it.

Run it

Open `WorldAware.unity` and press Play. No model needed — the sample shows what the model *would* be given, and lets you fire the action yourself.

What to notice

Perception picks the nouns; state picks the verbs; neither knows the other exists.

- The top box is the perception block: what she is told is nearby, in her own language, rebuilt every frame from what is actually in range.
- The bottom box is the `target` line of the grammar the model is constrained by. Those two lists are built from one recompute, so she cannot be told about an object she is then forbidden to name — or worse, name one she was never told about.
- Walk the character away from an object (move it in the Scene view while playing) and watch it leave both lists at once.

Press **look at** on any object. That goes through the *dispatcher*, not straight to the head — the same path a model's reply takes, so the button exercises the real contract rather than a shortcut past it.

The mistake this sample encodes

Select the target of `look_at` by casting to `ActionTargetBehaviour`, and it will never find the things perception registers, which are `AIPointOfInterest`. Both are targets; only one passes the cast. The verb sits in the grammar,

the model chooses it, the dispatcher delivers it, and nothing moves — and nothing logs, because as far as every component is concerned it worked.

Ask a target where it is with `action.Target.AimPointOf()`. It answers for either kind, and `null` is a real answer: an id may name a topic or a direction rather than a thing.