Guide Complet : Créer un Agent IA Vocal en Temps Réel avec Gemini Live API

Niveau débutant · Python · Édition enrichie

NIZAR SOILIHI | Septembre 2026

Ce guide vous accompagne pas à pas, de zéro jusqu'à un agent vocal fonctionnel : vous parlez dans votre micro, l'IA vous répond à voix haute, vous pouvez l'interrompre, et elle peut même appeler vos propres fonctions Python (vérifier une réservation, consulter un colis, etc.).

Aucune connaissance préalable en audio ou en intelligence artificielle n'est nécessaire. Chaque notion est expliquée simplement, avec un schéma ou un exemple. Seules des bases de Python sont utiles (variables, fonctions, `if`, boucles).

Comment utiliser ce guide ? Suivez les parties dans l'ordre. Chaque exemple de code est complet et peut être copié, collé et exécuté. Les blocs « À retenir » résument l'essentiel ; les blocs « Piège » signalent les erreurs classiques des débutants.

Note importante sur les noms de modèles. Les modèles Gemini Live évoluent vite. Au moment de la rédaction, les documentations officielles recommandent `gemini-3.8-live` pour la plupart des agents vocaux. Les anciens modèles (`gemini-2.0-flash-live-001`, `gemini-2.5-flash-native-audio-*`, `gemini-3.1-flash-live-preview`) fonctionnent encore mais suivent des règles légèrement différentes. Vérifiez toujours la page officielle des modèles avant de démarrer : https://ai.google.dev/gemini-api/docs/live.

Table des matières

Table des matières

Table des matières **2**
1. Comprendre le changement de paradigme **4**
1.1 L'ancienne méthode : le « talkie-walkie » 4
1.2 La nouvelle méthode : la conversation native 4
2. Le vocabulaire du débutant **5**
3. Comment fonctionne un agent vocal Gemini Live **6**
3.1 Vue d'ensemble 6
3.2 Les formats audio à respecter absolument 6
3.3 Le cycle d'un échange 6
4. Préparer son environnement **7**
4.1 Installer Python 7
4.2 Créer un environnement virtuel (recommandé) 7
4.3 Installer les bibliothèques 7
4.4 Obtenir une clé d'API Gemini 7
4.5 Stocker la clé dans un fichier `.env` 7
4.6 Tester la clé 8
4.7 Tester son micro et ses haut-parleurs 8
5. Premier test : dialoguer sans micro **8**
6. Construire un agent vocal complet (micro et haut-parleur) **9**
6.1 Le principe : trois tâches en parallèle 9
6.2 Le code complet 10
6.3 Comprendre les points clés 11
7. Les « superpouvoirs » d'un agent vocal **12**
7.1 L'interruption (barge-in) et la détection vocale (VAD) 12
7.2 Les transcriptions (voir ce qui est dit) 13
7.3 Choisir la voix et la langue 13
8. Donner des outils à l'agent (function calling) **13**
8.1 Le principe 14
8.2 Étape 1 : décrire l'outil à Gemini 14
8.3 Étape 2 : écrire la vraie fonction 15
8.4 Étape 3 : traiter le `tool_call` et répondre 15
8.5 Outils qui ne bloquent pas la conversation 15
8.6 Chercher sur Internet : la recherche Google 16
8.7 Bonnes pratiques pour les outils 16
9. Projet complet : la réceptionniste du restaurant « Le Gourmet » **16**
9.1 Scénario visé 16

9.2 Le code complet : `restaurant.py` 17
9.3 Comment tester méthodiquement 19
10. Écrire un bon prompt système pour la voix 19
10.1 Structure recommandée 19
10.2 Ce qui change à l'oral 19
10.3 Exemple : mauvais prompt et bon prompt 20
11. Sessions longues : compression et reprise 20
11.1 La compression de contexte 20
11.2 La reprise de session 20
12. Voir en plus d'entendre : envoyer la caméra 21
12.1 Installation 21
12.2 Envoyer les images de la webcam 21
12.3 Exemple de consignes pour un agent « qui voit » 21
13. Passer au web : navigateur, serveur relais et sécurité 22
13.1 Le serveur relais avec FastAPI 22
13.2 La page web : `static/index.html` 23
13.3 Où brancher les outils ? 24
13.4 Les alternatives « clés en main » 24
14. Autres cas d'usage concrets 25
14.1 Le coach d'apprentissage des langues 25
14.2 L'assistant de support technique visuel 25
14.3 L'agent de prise de rendez-vous médical (administratif) 25
14.4 Le guide vocal d'un musée ou d'une ville 25
14.5 L'entraîneur d'entretien d'embauche 25
15. Pièges courants et dépannage 26
15.1 Les cinq pièges classiques 26
15.2 Tableau de dépannage 26
15.3 Déboguer efficacement 26
16. Limites, coûts et bonnes pratiques 27
16.1 Limites à connaître 27
16.2 Coûts 27
16.3 Sécurité et confidentialité (checklist) 27
16.4 Bonnes pratiques de conception 27
17. Exercices et prochaines étapes 27
18. Ressources officielles 28
19. Annexe : corrections par rapport à la première version du guide 28

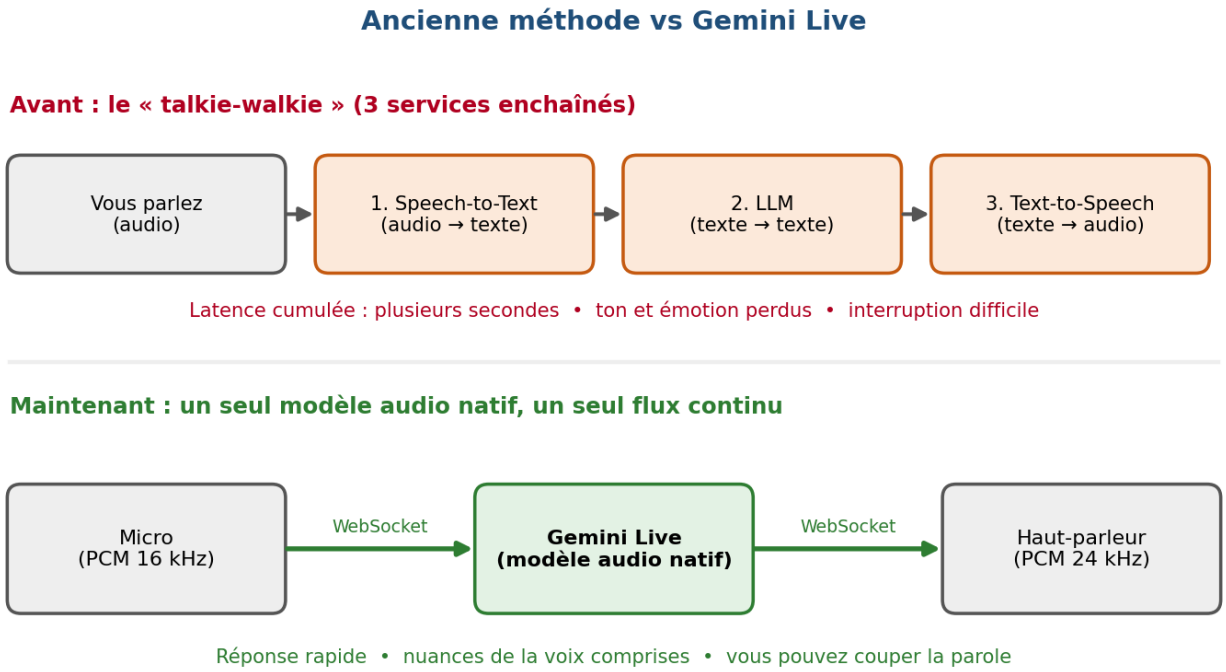
1. Comprendre le changement de paradigme

1.1 L'ancienne méthode : le « talkie-walkie »

Avant l'arrivée des modèles audio natifs, construire une IA vocale demandait d'enchaîner **trois services** différents :

1. **Speech-to-Text (STT)** : transforme votre voix en texte.
2. **LLM** (modèle de langage) : lit le texte et écrit une réponse en texte.
3. **Text-to-Speech (TTS)** : transforme la réponse écrite en voix.

Chaque étape doit attendre la fin de la précédente. C'est comme un talkie-walkie : chacun parle à son tour, et il faut dire « à vous » pour passer la main.



Ancienne méthode (STT, LLM, TTS) comparée à Gemini Live

Pourquoi ce système posait problème :

- **Latence élevée :** les délais de chaque service s'additionnent. Il fallait souvent attendre plusieurs secondes entre deux phrases.
- **Perte d'émotion :** le ton, le sarcasme, l'hésitation ou le stress dans la voix disparaissent dès la conversion en texte.
- **Interruption difficile :** l'IA ne vous « entend » pas pendant qu'elle lit sa réponse.

1.2 La nouvelle méthode : la conversation native

Gemini Live utilise un **modèle audio multimodal natif** : il reçoit de l'**audio** et produit de l'**audio**, dans **un seul flux continu** ouvert via une connexion **WebSocket** (un « tuyau » permanent entre votre programme et Google, par opposition à une requête classique qui s'ouvre et se ferme à chaque message).

Ce qui change concrètement :

Critère	Ancienne méthode (STT → LLM → TTS)	Gemini Live (audio natif)
Nombre de services	3 à assembler	1 seul
Délai de réponse	Plusieurs secondes	Très court, conversation fluide
Ton et émotion de la voix	Perdus	Compris par le modèle
Interruption	Difficile à programmer	Native (détection de voix intégrée)
Complexité du code	Élevée	Réduite

À retenir. Avec Gemini Live, l'IA écoute et parle en même temps, comme un interlocuteur humain. Vous n'avez plus à coller trois briques ensemble.

2. Le vocabulaire du débutant

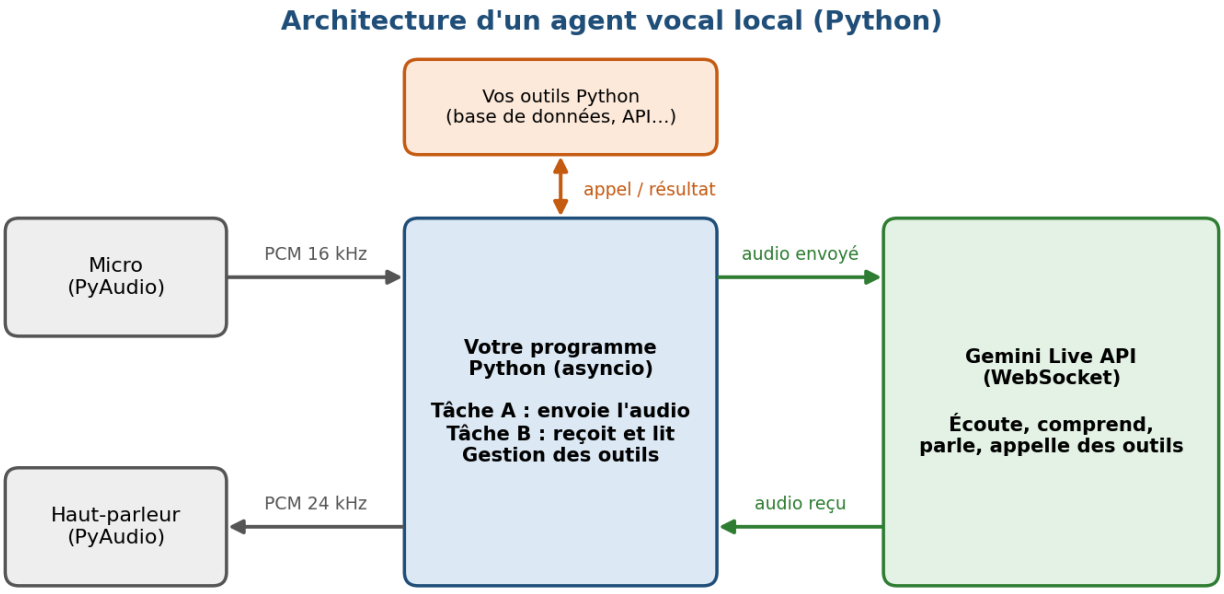
Avant de coder, voici les mots que vous allez rencontrer. Revenez à ce tableau quand un terme vous échappe.

Terme	Explication simple	Exemple
API	Une « porte d'entrée » qui permet à votre programme de demander un service à un autre système.	Votre code demande à Gemini de répondre.
Clé d'API	Un mot de passe personnel qui prouve que c'est bien vous qui faites la demande.	`AIza...` (à ne jamais partager).
WebSocket	Une connexion qui reste ouverte, dans les deux sens, pour échanger des données en continu.	Un appel téléphonique, contrairement à un SMS.
Session	Une conversation ouverte avec le modèle, du début à la fin.	Un appel entre l'ouverture et le raccroché.
PCM	Le format audio « brut », non compressé : une suite de nombres qui décrivent l'onde sonore.	Le contenu d'un fichier `.wav`, sans l'en-tête.
Fréquence d'échantillonnage (*sample rate*)	Nombre de mesures du son par seconde. Plus il est élevé, plus le son est fin.	16 000 Hz = 16 000 mesures par seconde.
Chunk (morceau)	Un petit paquet d'audio envoyé ou reçu, de quelques dizaines de millisecondes.	1024 échantillons envoyés à la suite.
VAD (*Voice Activity Detection*)	Détection d'activité vocale : le système repère quand un humain commence ou arrête de parler.	Permet de savoir que vous coupez la parole.
Barge-in	Le fait d'interrompre l'IA pendant qu'elle parle.	« Attends, dis-moi juste le prix ! »
Tool calling / function calling	L'IA demande à votre programme d'exécuter une fonction pour obtenir une information ou faire une action.	`verifier_colis("1234")`.
Prompt système (*system instruction*)	Les consignes de départ qui définissent le rôle et la personnalité de l'IA.	« Tu es un assistant de réservation poli. »
asyncio	Le module Python qui permet de faire plusieurs choses « en même temps » (écouter et parler).	Deux tâches qui tournent en parallèle.

3. Comment fonctionne un agent vocal Gemini Live

3.1 Vue d'ensemble

Un agent vocal local se compose de quatre éléments :



Architecture d'un agent vocal local en Python

1. **Le micro** capture votre voix (via la bibliothèque PyAudio).
2. **Votre programme Python** découpe l'audio en petits morceaux et les envoie à Gemini (tâche A), en même temps qu'il reçoit et joue les réponses (tâche B).
3. **Gemini Live API** écoute, comprend, décide de répondre ou d'appeler un outil, et renvoie de l'audio.
4. **Le haut-parleur** joue la voix de l'IA.

3.2 Les formats audio à respecter absolument

C'est la source d'erreurs n°1 des débutants. Retenez ces deux valeurs :

Sens	Format	Fréquence	Canaux
Vers Gemini (votre micro)	PCM brut 16 bits, little-endian	**16 000 Hz** (16 kHz)	Mono
Depuis Gemini (sa voix)	PCM brut 16 bits, little-endian	**24 000 Hz** (24 kHz)	Mono

Le type MIME à indiquer à l'envoi est : `audio/pcm;rate=16000`. Le `;rate=16000` est important : il précise la fréquence.

Piège. Si vous jouez la voix de Gemini (24 kHz) sur un flux configuré à 16 kHz, elle sonnera lente et grave. Si vous faites l'inverse, elle sonnera aiguë et accélérée (« voix de chipmunk »).

3.3 Le cycle d'un échange

1. Vous ouvrez une **session** (connexion WebSocket) avec un modèle et une configuration.
2. Votre programme **envoie en continu** l'audio du micro.
3. Gemini détecte que vous avez fini de parler et **répond** en audio (par petits morceaux, en continu).
4. Si vous reparlez pendant sa réponse, il envoie un signal `interrupted` : vous devez alors vider la file d'attente audio.

5. S'il a besoin d'une information externe, il envoie un `tool_call` : vous exécutez la fonction et renvoyez le résultat.
6. À la fin, vous fermez la session.

4. Préparer son environnement

4.1 Installer Python

Il vous faut **Python 3.10 ou plus récent**. Vérifiez votre version :

```
python --version
```

Sur certains systèmes (Linux, macOS), la commande est `python3`.

4.2 Créer un environnement virtuel (recommandé)

Un environnement virtuel isole les bibliothèques de ce projet, pour éviter les conflits avec d'autres projets.

```
mkdir agent-vocal
cd agent-vocal
python -m venv .venv

# Activer l'environnement
source .venv/bin/activate        # Linux / macOS
.venv\Scripts\activate           # Windows (cmd ou PowerShell)
```

Vous devriez voir `(.venv)` au début de votre ligne de commande.

4.3 Installer les bibliothèques

```
pip install -U google-genai pyaudio python-dotenv
```

- `google-genai` : le SDK officiel (version 2.3.0 ou plus récente recommandée, avec l'installation `-U`).
- `pyaudio` : accès au micro et aux haut-parleurs.
- `python-dotenv` : lit votre clé API depuis un fichier `.env`.

Attention. N'utilisez pas l'ancien paquet `google-generativeai` : il est déprécié. Le bon paquet s'appelle **`google-genai`** (sans « erativeai »). Les modules `asyncio` et `websockets` n'ont pas besoin d'être installés à part : `asyncio` fait partie de Python et le SDK gère lui-même le WebSocket.

Si l'installation de PyAudio échoue :

Système	Solution
macOS	`brew install portaudio` puis `pip install pyaudio`
Ubuntu / Debian	`sudo apt-get install portaudio19-dev python3-dev` puis `pip install pyaudio`
Windows	`pip install pyaudio` fonctionne en général directement. En cas d'échec, installez une version précompilée (wheel) correspondant à votre version de Python.

4.4 Obtenir une clé d'API Gemini

1. Rendez-vous sur **Google AI Studio** : https://aistudio.google.com/.
2. Connectez-vous avec un compte Google.
3. Créez une clé d'API (**Get API key**).
4. Copiez-la, sans la partager ni la publier.

4.5 Stocker la clé dans un fichier `.env`

Créez un fichier nommé `.env` à la racine du projet :

```
GEMINI_API_KEY=votre_cle_api_ici
```

Sécurité. Ne mettez **jamais** votre clé directement dans le code, et n'envoyez jamais le fichier `.env` sur GitHub. Si vous utilisez Git, ajoutez la ligne `.env` dans un fichier `.gitignore`.

4.6 Tester la clé

Créez `test_cle.py` :

```
import os
from dotenv import load_dotenv
from google import genai

load_dotenv()  # lit le fichier .env

client = genai.Client(api_key=os.environ["GEMINI_API_KEY"])
reponse = client.models.generate_content(
    model="gemini-2.5-flash",
    contents="Dis bonjour en une phrase.",
)
print(reponse.text)
```

Lancez `python test_cle.py`. Si une phrase s'affiche, votre clé fonctionne. (Si le nom du modèle a changé, remplacez-le par un modèle texte disponible dans AI Studio.)

4.7 Tester son micro et ses haut-parleurs

Ce petit script liste les périphériques audio détectés. Utile pour vérifier que PyAudio « voit » votre micro :

```
import pyaudio

p = pyaudio.PyAudio()
for i in range(p.get_device_count()):
    info = p.get_device_info_by_index(i)
    entree = info["maxInputChannels"] > 0
    sortie = info["maxOutputChannels"] > 0
    print(f"[{i}] {info['name']}  entrée={entree}  sortie={sortie}")
p.terminate()
```

Notez le numéro du périphérique que vous voulez utiliser : vous pourrez le passer à `input_device_index` dans PyAudio.

> **Conseil.** Utilisez un **casque** pendant tout le développement. Sans casque, le micro capte la voix de l'IA et l'agent croit que vous l'interrompez (effet d'écho, voir la partie 15).

---

5. Premier test : dialoguer sans micro

Avant de brancher le micro (source de nombreux problèmes), vérifions que la connexion à Gemini Live fonctionne. Ce script envoie **une phrase écrite** et enregistre la **réponse vocale** dans un fichier `reponse.wav` que vous pourrez écouter.

Créez `premier_test.py` :

```
import asyncio
import os
import wave

from dotenv import load_dotenv
from google import genai
from google.genai import types

load_dotenv()
client = genai.Client(api_key=os.environ["GEMINI_API_KEY"])

MODELE = "gemini-3.8-live"  # vérifiez le nom actuel dans la documentation

config = types.LiveConnectConfig(
    response_modalities=[types.Modality.AUDIO],
    # Pour afficher aussi le texte de ce que dit l'IA :
    output_audio_transcription=types.AudioTranscriptionConfig(),
)
```

```python
async def main():
    async with client.aio.live.connect(model=MODELE, config=config) as session:
        print("Connecté à Gemini Live.")

        # On envoie une phrase écrite à l'IA
        await session.send_realtime_input(
            text="Bonjour ! Présente-toi en deux phrases courtes."
        )

        # On prépare un fichier .wav : mono, 16 bits, 24 000 Hz (voix de Gemini)
        with wave.open("reponse.wav", "wb") as wf:
            wf.setnchannels(1)
            wf.setsampwidth(2)        # 2 octets = 16 bits
            wf.setframerate(24000)    # la voix de Gemini est en 24 kHz

            # On lit les messages reçus jusqu'à la fin de la réponse
            async for reponse in session.receive():
                contenu = reponse.server_content
                if contenu is None:
                    continue

                if contenu.model_turn:
                    for partie in contenu.model_turn.parts:
                        if partie.inline_data:  # morceau d'audio
                            wf.writeframes(partie.inline_data.data)

                if contenu.output_transcription:
                    print(contenu.output_transcription.text, end="", flush=True)

                if contenu.turn_complete:
                    print("\nRéponse terminée.")
                    break


if __name__ == "__main__":
    asyncio.run(main())
```

Lancez-le : `python premier_test.py`, puis ouvrez `reponse.wav` avec n'importe quel lecteur audio.

Ce que fait ce code, en clair :

Ligne ou bloc	Rôle
`genai.Client(api_key=...)`	Crée le « téléphone » qui appelle Google avec votre clé.
`LiveConnectConfig(...)`	Prépare les réglages : ici, « réponds en audio ».
`client.aio.live.connect(...)`	Ouvre la session (le WebSocket). `async with` la ferme proprement à la fin.
`send_realtime_input(text=...)`	Envoie une entrée à l'IA (ici du texte, plus tard de l'audio).
`session.receive()`	Reçoit les messages de l'IA, l'un après l'autre.
`inline_data.data`	Un morceau d'audio brut (PCM 24 kHz) à écrire ou à jouer.
`turn_complete`	Signal : « j'ai fini ma réponse ».

> **À retenir.** Une session Live est une conversation : on **envoie** des entrées et on **reçoit** des messages. La voix arrive par petits morceaux, dans `server_content.model_turn.parts[].inline_data.data`.

6. Construire un agent vocal complet (micro et haut-parleur)

Passons au vrai agent : il écoute votre micro en continu et joue sa voix en direct, avec gestion de l'interruption.

6.1 Le principe : trois tâches en parallèle

Pour qu'une conversation soit fluide, trois choses doivent tourner **simultanément** :

1. **Envoyer** l'audio du micro à Gemini, sans jamais s'arrêter.
2. **Recevoir** les réponses de Gemini (audio, signaux d'interruption, appels d'outils).

3. **Jouer** l'audio reçu dans le haut-parleur, dans l'ordre.

On les relie par une **file d'attente** (`asyncio.Queue`) : la tâche « recevoir » y dépose les morceaux d'audio, la tâche « jouer » les en retire. Quand l'utilisateur interrompt l'IA, il suffit de **vider la file** pour que la voix s'arrête tout de suite.

Comme PyAudio est une bibliothèque « bloquante » (une lecture attend le son), on l'appelle via `asyncio.to_thread(...)` pour ne pas geler les autres tâches.

6.2 Le code complet

Créez `agent_vocal.py` :

```
import asyncio
import os

import pyaudio
from dotenv import load_dotenv
from google import genai
from google.genai import types

load_dotenv()
client = genai.Client(api key=os.environ["GEMINI API KEY"])

MODELE = "gemini-3.8-live"

# --- Réglages audio ----------------------------------------------------
FORMAT_AUDIO = pyaudio.paInt16   # 16 bits
CANAUX = 1                       # mono
FREQ_ENTREE = 16000              # micro -> Gemini : 16 kHz
FREQ_SORTIE = 24000              # Gemini -> haut-parleur : 24 kHz
TAILLE_CHUNK = 1024              # échantillons par paquet

# --- Réglages de l'agent -----------------------------------------------
CONSIGNES = (
    "Tu es un assistant vocal francophone, chaleureux et concis. "
    "Tu réponds en une à trois phrases courtes, comme dans une vraie "
    "conversation orale. Tu ne lis jamais de listes à puces à voix haute."
)

config = types.LiveConnectConfig(
    response_modalities=[types.Modality.AUDIO],
    speech_config=types.SpeechConfig(
        voice_config=types.VoiceConfig(
            prebuilt_voice_config=types.PrebuiltVoiceConfig(voice_name="Puck")
        )
    ),
    system_instruction=types.Content(parts=[types.Part(text=CONSIGNES)]),
    input_audio_transcription=types.AudioTranscriptionConfig(),
    output_audio_transcription=types.AudioTranscriptionConfig(),
)


async def main():
    pa = pyaudio.PyAudio()

    micro = pa.open(
        format=FORMAT_AUDIO, channels=CANAUX, rate=FREQ_ENTREE,
        input=True, frames_per_buffer=TAILLE_CHUNK,
    )
    haut_parleur = pa.open(
        format=FORMAT_AUDIO, channels=CANAUX, rate=FREQ_SORTIE, output=True,
    )

    file_lecture: asyncio.Queue = asyncio.Queue()

    async with client.aio.live.connect(model=MODELE, config=config) as session:
        print("Connecté ! Parlez dans votre micro (Ctrl+C pour arrêter).")

        # Tâche 1 : micro -> Gemini
        async def envoyer_audio():
            while True:
                # exception_on_overflow=False évite un plantage si on prend du retard
                donnees = await asyncio.to_thread(
                    micro.read, TAILLE_CHUNK, False
                )
```

```
                await session.send_realtime_input(
                    audio=types.Blob(data=donnees, mime_type="audio/pcm;rate=16000")
                )

        # Tâche 2 : Gemini -> file d'attente (et affichage du texte)
        async def recevoir():
            while True:
                async for reponse in session.receive():
                    contenu = reponse.server_content
                    if contenu is None:
                        continue

                    # a) morceaux d'audio à jouer
                    if contenu.model_turn:
                        for partie in contenu.model_turn.parts:
                            if partie.inline_data:
                                file_lecture.put_nowait(partie.inline_data.data)

                    # b) l'utilisateur a coupé la parole : on vide la file
                    if contenu.interrupted:
                        while not file_lecture.empty():
                            file_lecture.get_nowait()
                        print("\n[Interruption : l'IA s'arrête]")

                    # c) transcriptions (texte de ce qui a été dit)
                    if contenu.input_transcription:
                        print(f"Vous  : {contenu.input_transcription.text}")
                    if contenu.output_transcription:
                        print(f"Agent : {contenu.output_transcription.text}")

        # Tâche 3 : file d'attente -> haut-parleur
        async def jouer_audio():
            while True:
                morceau = await file_lecture.get()
                await asyncio.to_thread(haut_parleur.write, morceau)

        try:
            await asyncio.gather(envoyer_audio(), recevoir(), jouer_audio())
        finally:
            micro.stop_stream()
            micro.close()
            haut_parleur.stop_stream()
            haut_parleur.close()
            pa.terminate()


if __name__ == "__main__":
    try:
        asyncio.run(main())
    except KeyboardInterrupt:
        print("\nAu revoir !")
```

Lancez-le avec un casque branché : `python agent_vocal.py`, puis parlez. Essayez de couper la parole à l'IA pendant qu'elle répond.

6.3 Comprendre les points clés

Élément	Pourquoi c'est important
`asyncio.gather(...)`	Lance les trois tâches en même temps. Si l'une plante, les autres sont arrêtées.
`asyncio.to_thread(micro.read, ...)`	Exécute la lecture bloquante du micro dans un fil séparé, sans geler le programme.
`mime_type="audio/pcm;rate=16000"`	Indique à Gemini le format exact de l'audio envoyé.
`while True: async for ... in session.receive()`	`receive()` s'arrête à la fin de chaque réponse ; la boucle `while True` le relance pour la suivante.
`contenu.interrupted`	Signal envoyé quand la détection de voix a repéré que vous parlez. Sans le vider, l'IA continuerait de parler avec l'audio déjà reçu.
`try ... finally`	Libère proprement le micro et le haut-parleur, même en cas d'erreur.
`Part(text=...)`	Syntaxe actuelle du SDK pour créer du contenu texte.

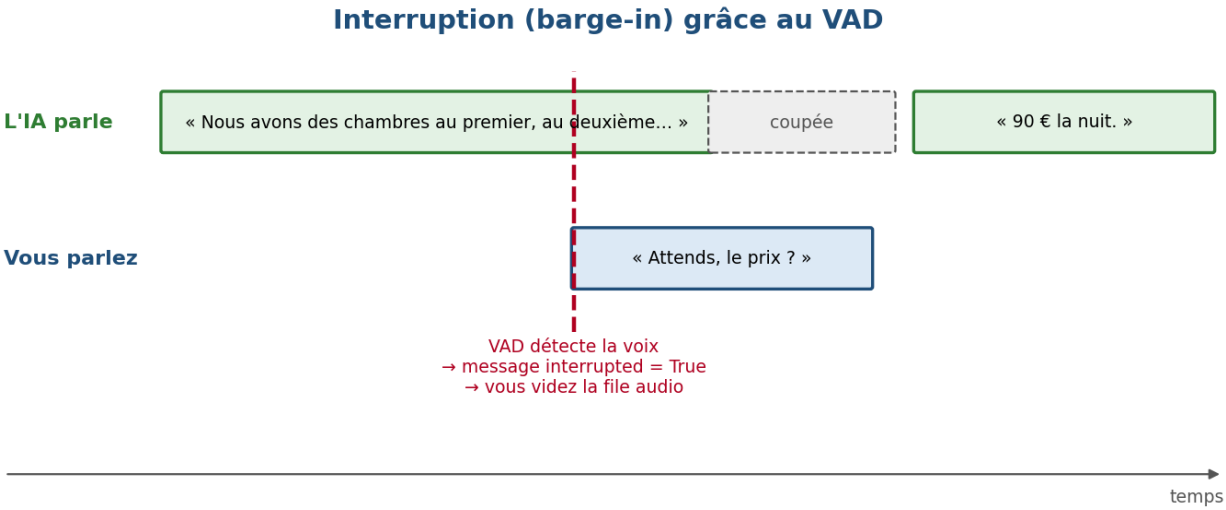
Piège. Traitez **toutes** les parties d'un message (`for partie in ...parts`) : un même message peut contenir plusieurs morceaux (audio et texte). Ne lisez pas seulement le premier.

7. Les « superpouvoirs » d'un agent vocal

7.1 L'interruption (barge-in) et la détection vocale (VAD)

Le **VAD** (*Voice Activity Detection*) permet au serveur de repérer quand un humain commence et arrête de parler. C'est ce qui rend possible l'échange suivant :

- *IA :* « Pour votre réservation, nous avons des chambres au premier étage, au deuxième… »
- *Vous :* « Attends, dis-moi juste le prix du premier ! »
- *IA (s'arrête aussitôt) :* « La chambre au premier étage est à 90 € la nuit. »



Interruption de l'IA grâce à la détection vocale

La détection est **activée par défaut**. Vous pouvez la régler si l'agent réagit trop vite (bruit de fond) ou trop lentement :

```
config = types.LiveConnectConfig(
    response_modalities=[types.Modality.AUDIO],
    realtime_input_config=types.RealtimeInputConfig(
        automatic_activity_detection=types.AutomaticActivityDetection(
            disabled=False,
            # Moins sensible au bruit : LOW. Plus réactif : HIGH.
            start_of_speech_sensitivity=types.StartSensitivity.START_SENSITIVITY_LOW,
            end_of_speech_sensitivity=types.EndSensitivity.END_SENSITIVITY_LOW,
            prefix_padding_ms=20,      # audio conservé avant le début de parole
            silence_duration_ms=500,   # silence nécessaire pour conclure « il a fini »
        )
    ),
)
```

Quelles valeurs choisir ?

Situation	Réglage conseillé
Environnement bruyant (rue, café)	Sensibilité de début de parole plus basse
Utilisateurs qui hésitent, réfléchissent en parlant	`silence_duration_ms` plus long (par exemple 800 à 1000)
Conversation très rapide	`silence_duration_ms` plus court (par exemple 300 à 500)

Note. Les noms exacts des paramètres peuvent évoluer avec les versions du SDK. En cas d'erreur, consultez la page « Live API capabilities » de la documentation officielle.

Mode « appuyer pour parler » (push-to-talk)

Pour certaines applications (talkie-walkie, borne bruyante), vous préférez décider vous-même quand l'utilisateur parle. On désactive alors la détection automatique et on signale le début et la fin :

```
config = types.LiveConnectConfig(
    response_modalities=[types.Modality.AUDIO],
    realtime_input_config=types.RealtimeInputConfig(
        automatic_activity_detection=types.AutomaticActivityDetection(disabled=True)
    ),
)

# Quand l'utilisateur appuie sur le bouton :
await session.send_realtime_input(activity_start=types.ActivityStart())
# ... on envoie l'audio avec send_realtime_input(audio=...) ...
# Quand l'utilisateur relâche le bouton :
await session.send_realtime_input(activity_end=types.ActivityEnd())
```

Et si vous coupez simplement le micro (bouton « muet »), pensez à prévenir le serveur que le flux audio est terminé :

```
await session.send_realtime_input(audio_stream_end=True)
```

7.2 Les transcriptions (voir ce qui est dit)

Pour afficher ou enregistrer les échanges sous forme de texte (sous-titres, historique, logs), activez les deux transcriptions dans la configuration :

```
config = types.LiveConnectConfig(
    response_modalities=[types.Modality.AUDIO],
    input_audio_transcription=types.AudioTranscriptionConfig(),   # ce que dit l'utilisateur
    output_audio_transcription=types.AudioTranscriptionConfig(),  # ce que dit l'IA
)
```

On les lit ensuite dans les messages reçus : `server_content.input_transcription.text` et `server_content.output_transcription.text` (voir le code de la partie 6).

Astuce. Une session ne peut renvoyer qu'**une seule modalité de réponse** : audio **ou** texte. Pour un agent vocal, choisissez `AUDIO` et activez la transcription de sortie si vous avez aussi besoin du texte.

7.3 Choisir la voix et la langue

La voix se règle avec `voice_name` (voir le code de la partie 6). Des voix comme `Puck`, `Charon`, `Kore`, `Fenrir` et `Aoede` sont disponibles, parmi d'autres. Le plus simple est de les écouter dans Google AI Studio et de choisir celle qui convient à votre personnage.

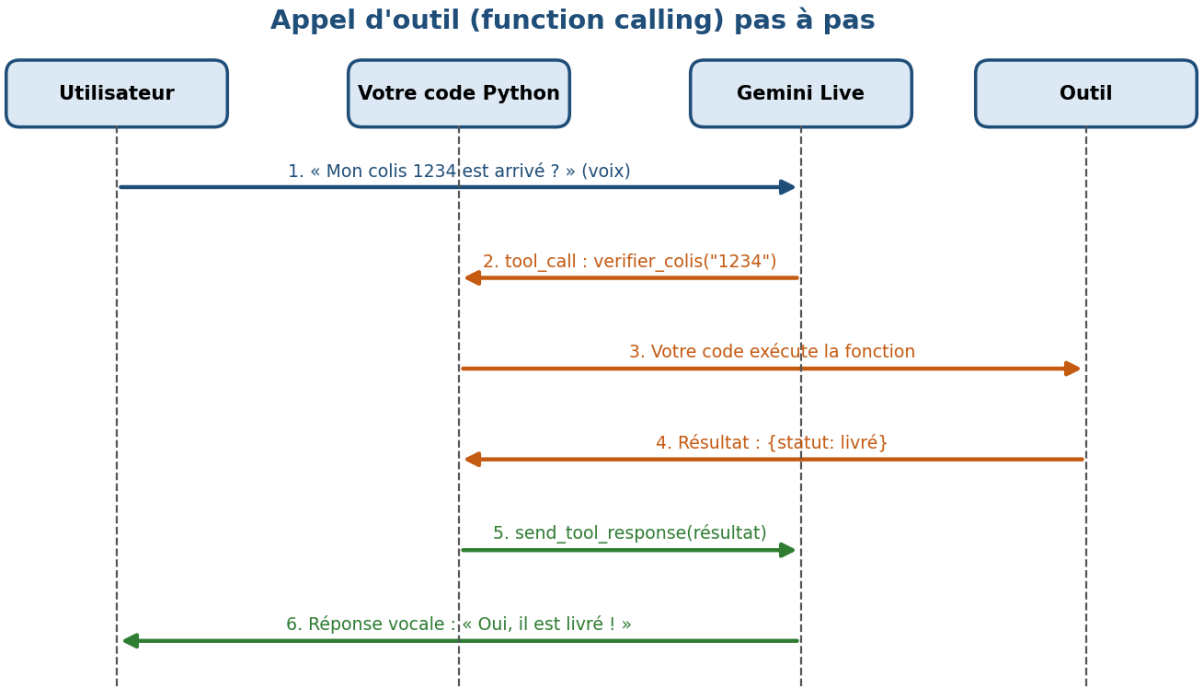
Les modèles natifs comprennent plus de 70 langues et peuvent changer de langue en cours de conversation. Pour éviter les surprises, dites clairement dans les consignes quelle langue utiliser :

```
Tu parles français par défaut. Si l'utilisateur te parle dans une autre
langue, réponds dans cette langue.
```

8. Donner des outils à l'agent (function calling)

C'est ici que l'agent devient vraiment utile : au lieu de simplement discuter, il peut **agir** ou **chercher une information** en appelant vos fonctions Python.

8.1 Le principe



Déroulement d'un appel d'outil

1. L'utilisateur pose une question qui demande une information externe.
2. Gemini n'invente pas la réponse : il envoie un message `tool_call` contenant le nom de la fonction et ses arguments.
3. **Votre code** exécute la vraie fonction Python.
4. Vous renvoyez le résultat avec `send_tool_response`.
5. Gemini formule la réponse à voix haute en s'appuyant sur ce résultat.

Très important. Contrairement à l'API texte classique, l'API Live **n'exécute jamais les fonctions à votre place**. C'est à vous de lire le `tool_call`, d'appeler la bonne fonction et de renvoyer la réponse manuellement.

8.2 Étape 1 : décrire l'outil à Gemini

Gemini ne « voit » pas votre code Python. Vous devez lui **décrire** chaque outil : son nom, à quoi il sert, et les paramètres attendus. Plus la description est claire, mieux l'IA saura quand l'utiliser.

```
DECLARATION_DISPONIBILITE = {
    "name": "verifier_disponibilite",
    "description": (
        "Vérifie si une table est disponible au restaurant. "
        "À utiliser dès que le client demande s'il reste de la place."
    ),
    "parameters": {
        "type": "object",
        "properties": {
            "heure": {
                "type": "string",
                "description": "Heure souhaitée au format 24 h, par exemple '19:30'.",
            },
            "personnes": {
                "type": "integer",
                "description": "Nombre de personnes à table.",
            },
        },
        "required": ["heure", "personnes"],
    },
}

TOOLS = [{"function_declarations": [DECLARATION_DISPONIBILITE]}]
```

On passe ensuite `tools=TOOLS` à la configuration (`LiveConnectConfig(..., tools=TOOLS)`).

8.3 Étape 2 : écrire la vraie fonction

```
def verifier_disponibilite(heure: str, personnes: int) -> dict:
    """Simule une base de données de réservations."""
    print(f"[OUTIL] Recherche d'une table pour {personnes} à {heure}")
    if heure.startswith("20"):
        return {"disponible": False, "message": "Complet à 20 h."}
    return {"disponible": True, "table": 42, "message": "Table disponible."}
```

Une fonction d'outil doit toujours renvoyer un **dictionnaire** (`dict`) simple : texte, nombres, booléens.

8.4 Étape 3 : traiter le `tool_call` et répondre

```
import asyncio
from google.genai import types

# Registre : nom de l'outil -> vraie fonction Python
OUTILS = {
    "verifier_disponibilite": verifier_disponibilite,
}


async def traiter_tool_call(session, tool_call):
    """Exécute chaque fonction demandée par Gemini et renvoie les résultats."""
    reponses = []
    for appel in tool_call.function_calls:
        fonction = OUTILS.get(appel.name)

        if fonction is None:
            resultat = {"erreur": f"Outil inconnu : {appel.name}"}
        else:
            try:
                arguments = dict(appel.args or {})
                # to_thread : évite de bloquer l'audio si la fonction est lente
                resultat = await asyncio.to_thread(fonction, **arguments)
            except Exception as erreur:  # ne jamais laisser planter la session
                resultat = {"erreur": str(erreur)}

        reponses.append(
            types.FunctionResponse(id=appel.id, name=appel.name, response=resultat)
        )

    await session.send_tool_response(function_responses=reponses)
```

Dans la tâche `recevoir()` de l'agent (partie 6), on ajoute la détection du `tool_call` **au même niveau** que `server_content` :

```
async def recevoir():
    while True:
        async for reponse in session.receive():
            # ... gestion de server_content (audio, interruption, texte) ...

            if reponse.tool_call:
                await traiter_tool_call(session, reponse.tool_call)
```

Piège. Recopiez bien `appel.id` dans la `FunctionResponse` : c'est lui qui permet à Gemini de relier votre résultat à sa demande.

8.5 Outils qui ne bloquent pas la conversation

Par défaut, quand un outil s'exécute, la conversation « attend » le résultat. Pour un outil lent (recherche, API distante), vous pouvez le déclarer **non bloquant** : l'utilisateur peut continuer à parler pendant que l'outil travaille.

```
DECLARATION_COLIS = {
    "name": "verifier_colis",
    "description": "Donne le statut d'un colis à partir de son numéro.",
    "behavior": "NON_BLOCKING",   # l'outil tourne en arrière-plan
    "parameters": {
        "type": "object",
        "properties": {"numero": {"type": "string", "description": "Numéro du colis"}},
        "required": ["numero"],
    },
}
```

Pour un outil non bloquant, vous indiquez dans la réponse **quand** l'IA doit en parler :

```
types.FunctionResponse(
    id=appel.id,
    name=appel.name,
    response={
        "statut": "livré ce matin",
        "scheduling": "WHEN_IDLE",   # ou "INTERRUPT" ou "SILENT"
    },
)
```

Valeur de `scheduling`	Effet
`INTERRUPT`	L'IA s'interrompt et annonce le résultat tout de suite.
`WHEN_IDLE`	L'IA attend d'avoir fini sa phrase en cours, puis l'annonce.
`SILENT`	L'IA retient l'information pour plus tard, sans en parler d'elle-même.

Note. Selon le modèle, l'exécution asynchrone peut être le comportement par défaut (c'est le cas des modèles récents `gemini-3.8-live`) ou une option à activer. Vérifiez la page « Tool use » de la documentation pour votre modèle.

8.6 Chercher sur Internet : la recherche Google

Sans écrire de fonction, vous pouvez donner à l'agent la recherche Google pour répondre à des questions d'actualité :

```
config = types.LiveConnectConfig(
    response_modalities=[types.Modality.AUDIO],
    tools=[{"google_search": {}}],
)
```

On peut combiner plusieurs outils : `tools=[{"google_search": {}}, {"function_declarations": [...]}]`.

8.7 Bonnes pratiques pour les outils

- **Nommez clairement** : `verifier_colis` vaut mieux que `func1`.
- **Décrivez précisément** quand utiliser l'outil dans le champ `description`.
- **Gardez les outils rapides** : au-delà d'une seconde, la conversation semble saccadée. Si nécessaire, passez en mode non bloquant.
- **Renvoyez des erreurs claires** (`{"erreur": "..."}`) plutôt que de laisser planter le programme : l'IA saura expliquer le problème à l'utilisateur.
- **Vérifiez les arguments** avant d'agir. Un utilisateur peut dire n'importe quoi, et l'IA peut se tromper. Ne donnez jamais à un outil le pouvoir de faire une action irréversible (paiement, suppression) sans confirmation.

9. Projet complet : la réceptionniste du restaurant « Le Gourmet »

Rassemblons tout ce que nous avons vu dans un seul agent : voix, interruption, transcription et **deux outils** (vérifier la disponibilité, puis réserver). Les « bases de données » sont simulées par un dictionnaire Python ; remplacez-les plus tard par une vraie base (SQLite, PostgreSQL, API).

9.1 Scénario visé

- *Client :* « Bonjour, avez-vous une table pour quatre à 19 h 30 ? »
- *Agent :* (appelle `verifier_disponibilite`) « Oui, il reste une table. Quel nom dois-je mettre pour la réservation ? »
- *Client :* « Amina. »
- *Agent :* « Parfait, je confirme : une table pour quatre à 19 h 30 au nom d'Amina. C'est bien cela ? »
- *Client :* « Oui. »
- *Agent :* (appelle `reserver_table`) « C'est réservé. À bientôt au Gourmet ! »

9.2 Le code complet : `restaurant.py`

```
import asyncio
import os

import pyaudio
from dotenv import load_dotenv
from google import genai
from google.genai import types

load_dotenv()
client = genai.Client(api_key=os.environ["GEMINI_API_KEY"])
MODELE = "gemini-3.8-live"

# ------------------------------------------------------------------ Données
CAPACITE_PAR_HEURE = 3          # nombre de tables réservables par créneau
RESERVATIONS: list[dict] = []   # simule une base de données


# ------------------------------------------------------------------ Outils
def verifier_disponibilite(heure: str, personnes: int) -> dict:
    """Indique s'il reste une table à l'heure demandée."""
    if personnes < 1 or personnes > 10:
        return {"disponible": False, "message": "Nous acceptons de 1 à 10 personnes."}
    prises = sum(1 for r in RESERVATIONS if r["heure"] == heure)
    if prises >= CAPACITE_PAR_HEURE:
        return {"disponible": False, "message": f"Complet à {heure}."}
    return {"disponible": True, "message": f"Il reste une table à {heure}."}


def reserver_table(heure: str, personnes: int, nom: str) -> dict:
    """Enregistre la réservation (après confirmation du client)."""
    dispo = verifier_disponibilite(heure, personnes)
    if not dispo["disponible"]:
        return {"reserve": False, "message": dispo["message"]}
    RESERVATIONS.append({"heure": heure, "personnes": personnes, "nom": nom})
    print(f"[RÉSERVATION] {nom}, {personnes} pers., {heure}")
    return {"reserve": True, "message": f"Réservation confirmée au nom de {nom}."}


OUTILS = {
    "verifier_disponibilite": verifier_disponibilite,
    "reserver_table": reserver_table,
}

DECLARATIONS = [
    {
        "name": "verifier_disponibilite",
        "description": "Vérifie s'il reste une table libre à une heure donnée.",
        "parameters": {
            "type": "object",
            "properties": {
                "heure": {"type": "string", "description": "Heure, format 24 h, ex. '19:30'"},
                "personnes": {"type": "integer", "description": "Nombre de personnes"},
            },
            "required": ["heure", "personnes"],
        },
    },
    {
        "name": "reserver_table",
        "description": (
            "Enregistre une réservation. À n'appeler QU'APRÈS avoir vérifié la "
            "disponibilité et obtenu la confirmation explicite du client."
        ),
        "parameters": {
            "type": "object",
            "properties": {
                "heure": {"type": "string", "description": "Heure, format 24 h"},
                "personnes": {"type": "integer", "description": "Nombre de personnes"},
                "nom": {"type": "string", "description": "Nom pour la réservation"},
            },
            "required": ["heure", "personnes", "nom"],
        },
    },
]

# ------------------------------------------------------------------ Consignes
CONSIGNES = """
# Rôle
Tu es Léa, la réceptionniste vocale du restaurant « Le Gourmet ».
```

```
# Style oral
- Tu parles français, avec chaleur et politesse.
- Phrases courtes (une à deux). Jamais de listes ni de symboles.
- Tu poses une seule question à la fois.

# Règles
- Pour toute demande de table, utilise d'abord verifier_disponibilite.
- Demande le nom du client, puis répète l'heure, le nombre de personnes et le nom.
- N'appelle reserver_table qu'après un « oui » clair du client.
- Si le restaurant est complet, propose un autre horaire.
- Tu ne connais rien d'autre que les réservations ; pour le reste, propose de
  rappeler le restaurant.
"""

config = types.LiveConnectConfig(
    response_modalities=[types.Modality.AUDIO],
    speech_config=types.SpeechConfig(
        voice_config=types.VoiceConfig(
            prebuilt_voice_config=types.PrebuiltVoiceConfig(voice_name="Kore")
        )
    ),
    system_instruction=types.Content(parts=[types.Part(text=CONSIGNES)]),
    tools=[{"function_declarations": DECLARATIONS}],
    input audio transcription=types.AudioTranscriptionConfig(),
    output_audio_transcription=types.AudioTranscriptionConfig(),
)

# ------------------------------------------------------------------ Audio
FORMAT_AUDIO, CANAUX = pyaudio.paInt16, 1
FREQ ENTREE, FREQ SORTIE, CHUNK = 16000, 24000, 1024


async def traiter_tool_call(session, tool_call):
    reponses = []
    for appel in tool_call.function_calls:
        fonction = OUTILS.get(appel.name)
        try:
            if fonction is None:
                resultat = {"erreur": f"Outil inconnu : {appel.name}"}
            else:
                resultat = await asyncio.to_thread(fonction, **dict(appel.args or {}))
        except Exception as erreur:
            resultat = {"erreur": str(erreur)}
        reponses.append(
            types.FunctionResponse(id=appel.id, name=appel.name, response=resultat)
        )
    await session.send_tool_response(function_responses=reponses)


async def main():
    pa = pyaudio.PyAudio()
    micro = pa.open(format=FORMAT_AUDIO, channels=CANAUX, rate=FREQ_ENTREE,
                    input=True, frames_per_buffer=CHUNK)
    haut_parleur = pa.open(format=FORMAT_AUDIO, channels=CANAUX,
                           rate=FREQ_SORTIE, output=True)
    file_lecture: asyncio.Queue = asyncio.Queue()

    async with client.aio.live.connect(model=MODELE, config=config) as session:
        print("Réceptionniste prête. Parlez ! (Ctrl+C pour quitter)")

        async def envoyer_audio():
            while True:
                donnees = await asyncio.to_thread(micro.read, CHUNK, False)
                await session.send_realtime_input(
                    audio=types.Blob(data=donnees, mime_type="audio/pcm;rate=16000")
                )

        async def recevoir():
            while True:
                async for reponse in session.receive():
                    contenu = reponse.server_content
                    if contenu:
                        if contenu.model_turn:
                            for partie in contenu.model_turn.parts:
                                if partie.inline_data:
                                    file_lecture.put_nowait(partie.inline_data.data)
                        if contenu.interrupted:
                            while not file_lecture.empty():
                                file_lecture.get_nowait()
```

```
                        if contenu.input_transcription:
                            print(f"Client : {contenu.input_transcription.text}")
                        if contenu.output_transcription:
                            print(f"Léa    : {contenu.output_transcription.text}")
                    if reponse.tool_call:
                        await traiter_tool_call(session, reponse.tool_call)

        async def jouer_audio():
            while True:
                morceau = await file_lecture.get()
                await asyncio.to_thread(haut_parleur.write, morceau)

        try:
            await asyncio.gather(envoyer_audio(), recevoir(), jouer_audio())
        finally:
            micro.close()
            haut_parleur.close()
            pa.terminate()


if __name__ == "__main__":
    try:
        asyncio.run(main())
    except KeyboardInterrupt:
        print("\nFin de service.")
```

9.3 Comment tester méthodiquement

1. Lancez `python restaurant.py` avec un casque.
2. Demandez une table pour deux à 19 h. Vous devez voir apparaître dans le terminal la transcription, puis la ligne `[RÉSERVATION]` après votre « oui ».
3. Répétez la demande pour la même heure **quatre fois** : la quatrième fois, l'agent doit annoncer que c'est complet (capacité de 3 par créneau).
4. Coupez la parole à l'agent au milieu d'une phrase : il doit se taire immédiatement.
5. Testez un cas hors sujet (« Quelle est la capitale du Japon ? ») : il doit rester dans son rôle.

Pour aller plus loin. Remplacez la liste `RESERVATIONS` par une base SQLite, ajoutez un outil `annuler_reservation`, et faites lire à l'agent les horaires d'ouverture depuis un fichier.

10. Écrire un bon prompt système pour la voix

Le **prompt système** (les consignes de départ) est ce qui donne sa personnalité à votre agent. Pour la voix, on n'écrit pas comme pour un chatbot texte : l'utilisateur **écoute**, il ne lit pas.

10.1 Structure recommandée

```
# Rôle
Qui est l'agent, pour qui travaille-t-il ?

# Personnalité et style oral
Ton, longueur des phrases, niveau de langage.

# Règles de conversation
Ce qu'il doit faire, ce qu'il ne doit jamais faire.

# Outils
Quand utiliser chaque outil, et dans quel ordre.

# Cas particuliers
Que faire si le client est en colère, si la demande est hors sujet, etc.
```

10.2 Ce qui change à l'oral

À faire	À éviter
Phrases courtes (une à trois)	Longs paragraphes
Une seule question à la fois	Enchaîner cinq questions
Chiffres et heures dits naturellement (« dix-neuf heures trente »)	Symboles, tableaux, puces, emojis

Reformuler pour confirmer une information importante	Supposer que tout a été compris
Consigne explicite de langue	Laisser l'IA deviner la langue

10.3 Exemple : mauvais prompt et bon prompt

Mauvais :

```
Tu es un assistant. Aide les gens.
```

Bon :

```
# Rôle
Tu es Karim, assistant vocal de la médiathèque municipale.

# Style oral
Tu parles français, avec un ton calme et amical. Tu réponds en deux phrases
maximum. Tu ne lis jamais de liste : tu proposes au plus deux choix à la fois.

# Règles
- Pour connaître la disponibilité d'un livre, utilise l'outil chercher_livre.
- Si tu ne sais pas, dis-le simplement et propose de contacter un bibliothécaire.
- N'invente jamais un titre, un auteur ou un horaire.
```

À retenir. Testez votre prompt en le **parlant vraiment**, pas seulement en le lisant : une réponse qui paraît correcte à l'écrit peut sembler longue et robotique à l'oral.

11. Sessions longues : compression et reprise

Une session Live n'est pas illimitée. Retenez ces ordres de grandeur (à confirmer dans la documentation actuelle de votre modèle) :

Contrainte	Valeur indicative
Session audio seule, sans compression	environ 15 minutes
Session audio + vidéo, sans compression	environ 2 minutes
Durée d'une connexion WebSocket	environ 10 minutes (avant fermeture par le serveur)
Fenêtre de contexte	environ 128 000 jetons en entrée

Pour tenir plus longtemps, il y a deux mécanismes complémentaires.

11.1 La compression de contexte

Elle fait « glisser » la mémoire de la conversation : les plus anciens échanges sont résumés ou supprimés pour laisser de la place aux nouveaux.

```
config = types.LiveConnectConfig(
    response_modalities=[types.Modality.AUDIO],
    context_window_compression=types.ContextWindowCompressionConfig(
        sliding window=types.SlidingWindow()
    ),
)
```

11.2 La reprise de session

Le serveur peut fermer la connexion (durée maximale, réseau instable). La **reprise de session** permet de se reconnecter **sans perdre la conversation**. Le serveur vous envoie un identifiant (*handle*) ; vous le conservez et le redonnez à la reconnexion.

```
handle = None  # dernier identifiant reçu

def creer_config(handle):
    return types.LiveConnectConfig(
        response_modalities=[types.Modality.AUDIO],
        context_window_compression=types.ContextWindowCompressionConfig(
```

```
            sliding_window=types.SlidingWindow()
        ),
        session_resumption=types.SessionResumptionConfig(handle=handle),
    )

# Dans la boucle de réception :
async for reponse in session.receive():
    maj = reponse.session_resumption_update
    if maj and maj.resumable and maj.new_handle:
        handle = maj.new_handle          # à conserver

    if reponse.go_away:                  # « la connexion va bientôt se fermer »
        print(f"Fermeture dans {reponse.go_away.time_left}, reconnexion à prévoir.")
```

Une boucle de reconnexion simple : quand la connexion se ferme, on relance `client.aio.live.connect(..., config=creer_config(handle))` avec le dernier `handle` connu.

À retenir. Pour un agent qui doit tourner plus de quelques minutes (borne, standard téléphonique), **activez toujours** la compression et la reprise de session.

12. Voir en plus d'entendre : envoyer la caméra

Gemini Live est **multimodal** : en plus du son, vous pouvez lui envoyer des **images** (au maximum environ **une image par seconde**), et il en tient compte dans la conversation.

12.1 Installation

```
pip install opencv-python
```

12.2 Envoyer les images de la webcam

```
import asyncio

import cv2
from google.genai import types


async def envoyer_video(session, index_camera=0):
    """Envoie une image JPEG par seconde depuis la webcam."""
    camera = cv2.VideoCapture(index camera)
    try:
        while True:
            ok, image = await asyncio.to_thread(camera.read)
            if not ok:
                break
            # Réduire la taille allège la connexion
            image = cv2.resize(image, (640, 480))
            ok, jpeg = cv2.imencode(".jpg", image)
            if ok:
                await session.send_realtime_input(
                    video=types.Blob(data=jpeg.tobytes(), mime type="image/jpeg")
                )
            await asyncio.sleep(1.0)      # 1 image par seconde
    finally:
        camera.release()
```

Ajoutez `envoyer_video(session)` dans le `asyncio.gather(...)` de votre agent, à côté des trois autres tâches.

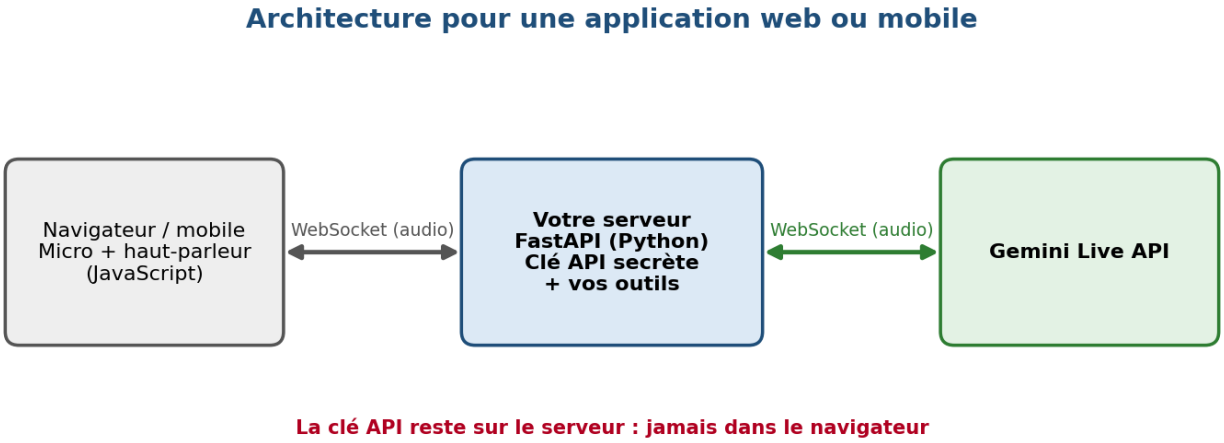
Attention. Les sessions avec vidéo sont beaucoup plus courtes sans compression (voir partie 11) et consomment davantage. N'envoyez la vidéo que lorsqu'elle est utile.

12.3 Exemple de consignes pour un agent « qui voit »

```
Tu es un assistant de dépannage. L'utilisateur te montre son appareil avec
sa caméra. Décris ce que tu vois avant de donner une instruction. Donne UNE
seule étape à la fois, puis attends que l'utilisateur confirme avant de
continuer. Si l'image est floue, demande de rapprocher ou d'éclairer l'appareil.
```

13. Passer au web : navigateur, serveur relais et sécurité

Jusqu'ici, tout tourne sur **votre ordinateur**. Pour une vraie application (site web, application mobile), l'utilisateur parle dans **son** navigateur. Deux architectures existent.



Architecture web avec serveur relais

Architecture	Principe	Quand l'utiliser
Serveur relais (recommandé au début)	Le navigateur parle à **votre serveur**, qui parle à Gemini.	Vous voulez garder la clé secrète et contrôler les outils côté serveur.
Connexion directe	Le navigateur parle directement à Gemini avec un **jeton éphémère** (temporaire).	Vous voulez réduire la latence et la charge serveur ; nécessite un peu plus de mise en place.

Règle d'or. Ne placez **jamais** votre clé d'API dans le code JavaScript d'un site : tout visiteur pourrait la copier. Passez par un serveur, ou utilisez des **jetons éphémères** générés par votre serveur (voir les ressources de la partie 18).

13.1 Le serveur relais avec FastAPI

Installation :

```
pip install fastapi "uvicorn[standard]"
```

Créez `serveur.py`. Le navigateur envoie l'audio micro (PCM 16 kHz) en binaire ; le serveur le transmet à Gemini et renvoie l'audio de la réponse (PCM 24 kHz) en binaire, plus de petits messages JSON pour les événements.

```
import asyncio
import os

from dotenv import load_dotenv
from fastapi import FastAPI, WebSocket, WebSocketDisconnect
from fastapi.staticfiles import StaticFiles
from google import genai
from google.genai import types

load_dotenv()
client = genai.Client(api_key=os.environ["GEMINI_API_KEY"])
MODELE = "gemini-3.8-live"

config = types.LiveConnectConfig(
    response_modalities=[types.Modality.AUDIO],
    system_instruction=types.Content(parts=[types.Part(
        text="Tu es un assistant vocal francophone, bref et chaleureux.")]),
    output_audio_transcription=types.AudioTranscriptionConfig(),
)

app = FastAPI()
```

```
@app.websocket("/ws")
async def ws_agent(ws: WebSocket):
    await ws.accept()
    async with client.aio.live.connect(model=MODELE, config=config) as session:

        async def navigateur_vers_gemini():
            while True:
                audio = await ws.receive_bytes()          # PCM 16 kHz
                await session.send_realtime_input(
                    audio=types.Blob(data=audio, mime_type="audio/pcm;rate=16000")
                )

        async def gemini_vers_navigateur():
            while True:
                async for rep in session.receive():
                    c = rep.server_content
                    if not c:
                        continue
                    if c.model_turn:
                        for p in c.model_turn.parts:
                            if p.inline_data:
                                await ws.send_bytes(p.inline_data.data)  # PCM 24 kHz
                    if c.interrupted:
                        await ws.send_json({"type": "interrupted"})
                    if c.output_transcription:
                        await ws.send_json(
                            {"type": "texte", "texte": c.output_transcription.text}
                        )

        try:
            await asyncio.gather(navigateur_vers_gemini(), gemini_vers_navigateur())
        except WebSocketDisconnect:
            pass


# Sert la page index.html du dossier "static"
app.mount("/", StaticFiles(directory="static", html=True), name="static")
```

Lancement : `uvicorn serveur:app --reload`, puis ouvrez http://localhost:8000.

13.2 La page web : `static/index.html`

Le navigateur doit (1) capturer le micro et le convertir en PCM 16 bits à 16 kHz, (2) jouer l'audio 24 kHz reçu, (3) arrêter la lecture en cas d'interruption. Voici un squelette minimal et fonctionnel.

```
<!DOCTYPE html>
<html lang="fr">
<head><meta charset="utf-8"><title>Agent vocal</title></head>
<body>
  <button id="demarrer">Démarrer la conversation</button>
  <pre id="journal"></pre>

<script>
const journal = document.getElementById("journal");

// Processeur audio : convertit le micro (Float32) en PCM 16 bits
const workletCode = `
class Capture extends AudioWorkletProcessor {
  process(entrees) {
    const canal = entrees[0][0];
    if (canal) {
      const pcm = new Int16Array(canal.length);
      for (let i = 0; i < canal.length; i++) {
        const s = Math.max(-1, Math.min(1, canal[i]));
        pcm[i] = s < 0 ? s * 0x8000 : s * 0x7FFF;
      }
      this.port.postMessage(pcm.buffer, [pcm.buffer]);
    }
    return true;
  }
}
registerProcessor("capture", Capture);`;

document.getElementById("demarrer").onclick = async () => {
  const ws = new WebSocket(`ws://${location.host}/ws`);
  ws.binaryType = "arraybuffer";
```

```
  // --- Micro : contexte à 16 kHz ---
  const ctxEntree = new AudioContext({ sampleRate: 16000 });
  const url = URL.createObjectURL(new Blob([workletCode], { type: "text/javascript" }));
  await ctxEntree.audioWorklet.addModule(url);
  const flux = await navigator.mediaDevices.getUserMedia({
    audio: { echoCancellation: true, noiseSuppression: true },
  });
  const source = ctxEntree.createMediaStreamSource(flux);
  const capture = new AudioWorkletNode(ctxEntree, "capture");
  source.connect(capture);
  capture.port.onmessage = (e) => {
    if (ws.readyState === WebSocket.OPEN) ws.send(e.data);
  };

  // --- Lecture : contexte à 24 kHz ---
  const ctxSortie = new AudioContext({ sampleRate: 24000 });
  let prochainInstant = 0;
  let sources = [];

  function jouer(buffer) {
    const pcm = new Int16Array(buffer);
    const audio = ctxSortie.createBuffer(1, pcm.length, 24000);
    const canal = audio.getChannelData(0);
    for (let i = 0; i < pcm.length; i++) canal[i] = pcm[i] / 32768;
    const src = ctxSortie.createBufferSource();
    src.buffer = audio;
    src.connect(ctxSortie.destination);
    prochainInstant = Math.max(prochainInstant, ctxSortie.currentTime);
    src.start(prochainInstant);
    prochainInstant += audio.duration;
    sources.push(src);
    src.onended = () => (sources = sources.filter((s) => s !== src));
  }

  function arreterLecture() {           // appelé en cas d'interruption
    sources.forEach((s) => { try { s.stop(); } catch (e) {} });
    sources = [];
    prochainInstant = 0;
  }

  ws.onmessage = (e) => {
    if (e.data instanceof ArrayBuffer) return jouer(e.data);
    const msg = JSON.parse(e.data);
    if (msg.type === "interrupted") arreterLecture();
    if (msg.type === "texte") journal.textContent += msg.texte;
  };
};
</script>
</body>
</html>
```

À savoir :

- Le navigateur n'autorise l'accès au micro que sur `localhost` ou en **HTTPS**. En production, utilisez `wss://` (WebSocket sécurisé) et un certificat HTTPS.
- `echoCancellation: true` demande au navigateur de supprimer l'écho : très utile, car l'utilisateur n'a pas toujours de casque.
- Ce squelette est volontairement minimal. En production, ajoutez la gestion des erreurs, un bouton d'arrêt, et l'authentification de vos utilisateurs.

13.3 Où brancher les outils ?

Toujours **côté serveur** (dans `gemini_vers_navigateur`), exactement comme dans la partie 8 : vous détectez `rep.tool_call`, exécutez la fonction, puis appelez `session.send_tool_response(...)`. Le navigateur n'a jamais accès à vos bases de données ni à vos secrets.

13.4 Les alternatives « clés en main »

Si vous ne voulez pas gérer vous-même le son, le réseau et la téléphonie, des frameworks existent et s'intègrent à Gemini Live : **LiveKit Agents**, **Pipecat**, **Firebase AI Logic**, et d'autres. Ils sont pratiques pour des agents téléphoniques ou des applications mobiles. Leurs liens sont dans la partie 18.

14. Autres cas d'usage concrets

14.1 Le coach d'apprentissage des langues

Idée : un tuteur qui converse avec l'élève, repère ses fautes de prononciation ou de grammaire, et corrige avec bienveillance.

```
# Rôle
Tu es Sam, tuteur d'anglais pour des francophones de niveau débutant (A2).

# Méthode
- Tu parles surtout en anglais, avec des phrases simples et lentes.
- Quand l'élève fait une faute, tu ne l'interromps pas : tu reformules
  correctement la phrase dans ta réponse, puis tu expliques en une phrase
  courte en français si nécessaire.
- Tu termines chaque échange par une question simple pour relancer.
- Toutes les cinq répliques, tu proposes un mini-défi de prononciation.
- Tu encourages toujours, sans exagérer.
```

Outil utile : `enregistrer_erreur(type, phrase, correction)` pour garder un carnet des fautes, puis un outil `resume_seance()` pour dresser un bilan à la fin.

14.2 L'assistant de support technique visuel

Idée : l'utilisateur pointe la caméra de son téléphone vers sa box Internet ou sa machine à café. L'agent voit l'appareil, écoute le problème et guide pas à pas.

- L'utilisateur : « Le voyant rouge clignote, que dois-je faire ? »
- L'agent (voit l'image) : « Je vois une box avec le voyant du milieu qui clignote en rouge. Pouvez-vous débrancher le câble d'alimentation noir, à gauche ? »

Ingrédients : envoi vidéo (partie 12), consignes « une étape à la fois » (partie 12.3), et un outil `chercher_manuel(modele, symptome)` qui interroge votre base de connaissances.

14.3 L'agent de prise de rendez-vous médical (administratif)

Un agent qui recueille le nom, le motif général et un créneau, puis appelle `creer_rendez_vous(...)`. Il ne doit **jamais donner de conseil médical**. Prévoyez dans les consignes :

```
Tu ne poses aucun diagnostic et ne donnes aucun conseil médical. En cas
d'urgence, tu demandes d'appeler immédiatement les secours.
```

14.4 Le guide vocal d'un musée ou d'une ville

L'agent connaît les œuvres ou lieux (chargés dans les consignes ou via un outil `chercher_oeuvre(nom)`), répond aux questions des visiteurs et s'adapte à leur langue. Combinez avec la recherche Google (partie 8.6) pour les informations récentes.

14.5 L'entraîneur d'entretien d'embauche

L'agent joue le recruteur, pose des questions, relance selon les réponses, puis donne un retour structuré à la fin (clarté, exemples, durée). Un outil `enregistrer_evaluation(criteres)` permet de conserver les résultats.

Comment choisir un premier projet ? Prenez un cas où le domaine est **étroit** (peu de sujets), les outils sont **simples** (deux ou trois fonctions), et où une erreur n'a **pas de conséquence grave**. Le restaurant, le coach de langue et le guide de musée sont de bons candidats.

15. Pièges courants et dépannage

15.1 Les cinq pièges classiques

1. **L'écho (effet Larsen).** Le micro reprend la voix de l'IA et l'agent croit être interrompu. *Solution :* portez un **casque** en développement ; sur le web, activez `echoCancellation`.
2. **Mauvaises fréquences audio.** Voix lente, grave ou aiguë. *Solution :* 16 kHz en entrée, 24 kHz en sortie, mono, 16 bits (partie 3.2).
3. **Outils trop lents.** Silences gênants pendant l'exécution. *Solution :* optimiser, ou déclarer l'outil `NON_BLOCKING` (partie 8.5).
4. **Ne traiter que la première partie d'un message.** Audio manquant. *Solution :* toujours boucler sur `parts`.
5. **Oublier de vider la file audio à l'interruption.** L'IA continue de parler quelques secondes après avoir été coupée. *Solution :* vider la file dès `interrupted` (partie 6).

15.2 Tableau de dépannage

Symptôme	Cause probable	Solution
`ModuleNotFoundError: No module named 'google'`	Bibliothèque non installée, ou mauvais environnement virtuel	Activez `.venv`, puis `pip install -U google-genai`
`pyaudio` ne s'installe pas	PortAudio manquant	Voir le tableau de la partie 4.3
`KeyError: 'GEMINI_API_KEY'`	Clé non chargée	Vérifiez le fichier `.env` et l'appel à `load_dotenv()`
Erreur d'authentification (401 / 403)	Clé invalide, révoquée ou sans accès au modèle	Recréez une clé dans AI Studio
Erreur « model not found »	Nom de modèle périmé ou mal orthographié	Consultez la liste actuelle des modèles Live
`AttributeError` sur `types.LiveModality`	Ancien exemple de code	Utilisez `types.Modality.AUDIO` (ou la chaîne `"AUDIO"`)
Aucun son en sortie	Mauvais périphérique de sortie, ou la file n'est pas lue	Vérifiez `jouer_audio()` et le périphérique (partie 4.7)
Voix de robot, lente ou accélérée	Fréquence d'échantillonnage incorrecte	24 000 Hz pour jouer la voix de Gemini
L'IA ne répond jamais	Micro muet, ou audio non envoyé	Affichez les transcriptions d'entrée pour vérifier ce qui est entendu
L'IA se coupe elle-même	Écho	Casque, ou baisse du volume
La session se ferme après quelques minutes	Limite de durée de connexion	Compression de contexte et reprise de session (partie 11)
L'outil n'est jamais appelé	Description vague, ou consignes contradictoires	Précisez `description` et le prompt système
Erreur 1007 ou fermeture inattendue du WebSocket	Message ou paramètre invalide pour ce modèle	Relisez les règles du modèle utilisé (`send_realtime_input`, un seul argument par appel)

15.3 Déboguer efficacement

- **Affichez tout** au début : `print(reponse)` dans la boucle de réception montre la structure exacte des messages.
- **Activez les transcriptions** : vous voyez ce que l'IA a réellement compris.
- **Testez sans micro** avec le script de la partie 5 pour isoler un problème de connexion d'un problème audio.
- **Changez une chose à la fois**, et gardez une version qui marche.

16. Limites, coûts et bonnes pratiques

16.1 Limites à connaître

- **Une seule modalité de réponse par session** : audio **ou** texte (pas les deux).
- **Durées limitées** sans compression et reprise (partie 11).
- **Certains outils ne sont pas disponibles** avec Live : l'exécution de code et le contexte d'URL, par exemple, ne sont pas pris en charge à ce jour. Vérifiez la page « Tool use ».
- **Les modèles évoluent** : noms, paramètres et comportements peuvent changer. Relisez la documentation avant un projet sérieux.

16.2 Coûts

L'audio en continu consomme des **jetons** (unités de facturation). Les tarifs varient selon le modèle et évoluent : consultez la page de tarification officielle avant de déployer. Réflexes simples pour limiter la facture :

- fermez la session quand l'utilisateur part ;
- n'envoyez la vidéo que si nécessaire ;
- évitez de laisser un agent « écouter » un micro ouvert sans utilisateur ;
- fixez un plafond de durée par conversation.

16.3 Sécurité et confidentialité (checklist)

- ☐ Clé d'API stockée dans `.env` ou un gestionnaire de secrets, jamais dans le code ni dans Git.
- ☐ Clé absente de tout code exécuté dans le navigateur.
- ☐ Jetons éphémères pour toute connexion directe navigateur → Gemini.
- ☐ Arguments des outils vérifiés (types, valeurs, limites) avant exécution.
- ☐ Action irréversible (paiement, suppression) soumise à une confirmation explicite.
- ☐ Utilisateurs prévenus qu'ils parlent à une IA, et informés de l'enregistrement éventuel des échanges.
- ☐ Données personnelles réduites au strict nécessaire dans les journaux (logs).
- ☐ Authentification et limitation du nombre de sessions par utilisateur.

16.4 Bonnes pratiques de conception

1. **Commencez petit** : un rôle, deux outils, un scénario.
2. **Écrivez pour l'oreille** : phrases courtes, une question à la fois.
3. **Prévoyez l'échec** : que dit l'agent si un outil plante ou si le réseau tombe ?
4. **Testez avec de vraies personnes** : accents, bruit de fond, hésitations.
5. **Journalisez les transcriptions** (avec consentement) pour améliorer le prompt.

17. Exercices et prochaines étapes

Niveau 1, découverte

1. Exécutez `premier_test.py` et changez la phrase envoyée. Écoutez le résultat.
2. Changez la voix (`Puck`, `Charon`, `Kore`…) et comparez.
3. Modifiez le prompt système pour que l'agent parle comme un guide touristique.

Niveau 2, interaction

4. Lancez `agent_vocal.py` avec un casque et testez l'interruption.
5. Ajoutez un outil `heure_actuelle()` qui renvoie l'heure locale, et demandez « Quelle heure est-il ? ».
6. Réglez la détection vocale pour un environnement bruyant (partie 7.1).

Niveau 3, projet

7. Étendez `restaurant.py` : ajoutez `annuler_reservation(nom)` et stockez les réservations dans SQLite.
8. Construisez le coach de langue (partie 14.1) avec un carnet d'erreurs.
9. Passez `restaurant.py` sur le web avec le serveur relais (partie 13).

Niveau 4, production

10. Ajoutez compression de contexte et reprise de session.
11. Ajoutez l'authentification et une limite de durée par utilisateur.
12. Mesurez le délai de réponse et améliorez les consignes.

Progression conseillée : texte vers audio (partie 5) → micro et haut-parleur (partie 6) → un outil (partie 8) → projet (partie 9) → web (partie 13) → production (parties 11 et 16).

18. Ressources officielles

Les liens suivants sont les références à consulter en priorité, car ils sont mis à jour par Google :

Sujet	Adresse
Présentation de la Live API	https://ai.google.dev/gemini-api/docs/live
Démarrer avec le SDK Google GenAI	https://ai.google.dev/gemini-api/docs/live-api/get-started-sdk
Capacités (voix, VAD, transcription)	https://ai.google.dev/gemini-api/docs/live-guide
Utilisation d'outils (function calling)	https://ai.google.dev/gemini-api/docs/live-tools
Gestion de session (compression, reprise)	https://ai.google.dev/gemini-api/docs/live-session
Jetons éphémères	https://ai.google.dev/gemini-api/docs/ephemeral-tokens
Google AI Studio (clés, essais)	https://aistudio.google.com/
Exemples de code (dépôt Google)	https://github.com/google-gemini/cookbook
Compétences de développement Live API	https://github.com/google-gemini/gemini-skills
LiveKit Agents (intégration Gemini)	https://docs.livekit.io/agents/models/realtime/plugins/gemini/
Pipecat (intégration Gemini Live)	https://docs.pipecat.ai/guides/features/gemini-live

19. Annexe : corrections par rapport à la première version du guide

La première version de ce guide contenait quelques éléments devenus incorrects ou risqués. Les voici, pour que vous ne les rencontriez pas dans d'autres tutoriels :

Ancienne version	Problème	Version corrigée
`pip install ... websockets asyncio`	`asyncio` est déjà dans Python ; le SDK gère le WebSocket	`pip install -U google-genai pyaudio python-dotenv`
`types.LiveModality.AUDIO`	Cette classe n'existe pas dans le SDK	`types.Modality.AUDIO` ou `"AUDIO"`
`session.send(input=types.LiveClientRealtimeInput(media_chunks=[...]))`	Ancienne méthode,	`session.send_realtime_input(audio=types.Blob(...))`

	remplacée	
`mime_type="audio/pcm"`	La fréquence n'est pas précisée	`"audio/pcm;rate=16000"`
`tools=[{"function_declarations": [verifier_disponibilite]}]` (fonction Python passée directement)	La Live API attend des **déclarations** (nom, description, paramètres), pas la fonction elle-même	Déclaration au format dictionnaire (partie 8.2)
Aucune gestion du `tool_call`	La Live API n'exécute pas les outils automatiquement	Lire `tool_call`, exécuter, puis `send_tool_response` (partie 8.4)
`types.Part.from_text("...")`	Signature qui varie selon les versions	`types.Part(text="...")`
`interrupted` détecté sans vider l'audio	L'IA continuait de parler	Vider la file de lecture (partie 6)
Lecture audio directement dans la boucle de réception	Bloque la réception pendant la lecture	File d'attente et tâche de lecture séparée
Modèle `gemini-2.0-flash-exp`	Modèle expérimental ancien	Modèle actuel recommandé (ex. `gemini-3.8-live`) : vérifiez la documentation
« L'IA répond en moins de 500 ms »	Chiffre non garanti, variable selon réseau et modèle	Latence « très courte », à mesurer dans votre contexte

---

Fin du guide. Bon développement, et n'hésitez pas à commencer par le plus petit exemple qui fonctionne : un agent qui dit bonjour est déjà un grand pas.