Interface à l'interpréteur

- Discussions préalables

Discussions préalables

Discussion Mistral :

Je comprends parfaitement ! À ce stade, il est crucial de **clarifier les cas d'usage** pour affiner l'architecture et éviter les choix techniques prématurés. Voici une synthèse des **scénarios clés** à valider, organisés par priorité et impact sur la conception. Ces cas d'usage serviront de base pour les décisions architecturales finales.

---

1. Cas d'Usage Principaux (Priorité Haute)

A. Utilisation en Classe (Collaboratif)

Scénario :

- **5 à 10 enfants** travaillent simultanément sur leurs postes (tablettes/PC).
- Ils **partagent des morceaux de code** (ex. : une procédure `POUR MAISON`) et voient les modifications en temps réel.
- L'enseignant peut **projeté la tortue virtuelle** sur un mur/écran pour que tous voient le résultat collectif.

Questions clés :

1. **Partage de code** :
   - Les enfants peuvent-ils **modifier le même code simultanément** (comme un Google Doc) ? → *Si oui* : Nécessite un système de synchronisation fine (CRDT ou WebSockets + verrouillage). → *Si non* : Un système de "copier-coller partagé" suffit (plus simple).
   - Faut-il un **historique des modifications** (pour revenir en arrière) ?
2. **Projection centrale** :
   - La tortue projetée doit-elle **afficher qui a envoyé la dernière commande** (ex. : "Alice a fait AVANCE 50") ?
   - Faut-il un **mode "présentation"** où l'enseignant peut geler l'affichage pour expliquer un concept ?
3. **Rôle de l'enseignant** :
   - Peut-il **bloquer l'envoi de commandes** pendant une explication ?
   - Peut-il **sauvegarder les projets** de tous les enfants en une fois ?

B. Intégration avec la Tortue-Robot

Scénario :

- La classe dispose d'**1 ou 2 tortues-robot** (MicroPython) qui se déplacent sur une feuille au sol.
- Les enfants envoient des commandes depuis leurs postes, et la tortue-robot **exécute les mêmes mouvements** que la tortue virtuelle projetée.

Questions clés :

1. **Synchronisation virtuel/physique** :
   - La tortue-robot doit-elle **suivre exactement** la tortue virtuelle (même position, même vitesse) ? → *Si oui* : Nécessite un système de calibration précis (ex. : 1 pas virtuel = 1 cm physique). → *Si non* : La tortue-robot exécute les commandes **indépendamment**, avec ses propres contraintes (ex. : elle bute sur un obstacle).
2. **Feedback physique** :
   - Si la tortue-robot est bloquée, faut-il **afficher un message d'erreur** sur tous les écrans (ex. : "La tortue ne peut pas avancer !") ?
   - Peut-on **envoyer des commandes "spéciales"** à la tortue-robot (ex. : allumer une LED, émettre un son) ?
3. **Logistique** :
   - Combien de tortues-robot seront utilisées **simultanément** ?
   - Faut-il prévoir un **système de réservation** (ex. : "C'est au tour de Bob d'utiliser la tortue") ?

C. Utilisation Individuelle (À la Maison)

Scénario :

- Un enfant utilise l'interpréteur **seul sur son poste**, sans tortue-robot ni projection.
- Il peut **sauvegarder ses projets** et les retrouver en classe.

Questions clés :

1. **Sauvegarde** :
   - Les projets sont-ils sauvegardés **localement** (fichier sur l'ordinateur) ou **sur un serveur** (compte utilisateur) ?
   - Faut-il un système de **synchronisation** entre la maison et l'école ?
2. **Compatibilité** :
   - L'interface doit-elle fonctionner sur **mobile/tablette** (ex. : iPad) ?
   - Faut-il une **version hors ligne** (pour les zones sans internet) ?

---

2. Cas d’Usage Secondaires (Priorité Moyenne)

A. Collaboration À Distance

Scénario :

- Deux classes (ou un enfant à la maison et un autre en classe) collaborent sur le même projet.

Questions :

- Faut-il un système d’**invitation par lien** (comme Google Docs) ?
- Comment gérer les **décalages horaires** ou les connexions instables ?

B. Extensions Pédagogiques

Scénario :

- L’enseignant veut **créer des défis** (ex. : "Dessine un hexagone") ou **limiter les commandes disponibles** pour les débutants.

Questions :

- Faut-il un **mode "défi"** où certaines commandes sont désactivées ?
- Peut-on **enregistrer des séquences** pour les rejouer plus tard (ex. : pour montrer une solution) ?

C. Accessibilité

Scénario :

- Un enfant malvoyant ou avec des troubles dys utilise l’outil.

Questions :

- Faut-il prévoir un **mode haute contraste** ou une **saisie vocale** ?
- Peut-on **agrandir la taille du texte** ou des boutons ?

---

3. Cas d’Usage Exclus (Priorité Basse)

- **Intégration avec d’autres outils** (ex. : exporter un dessin vers un logiciel de dessin).
- **Compétitions entre classes** (ex. : "Qui dessine le plus beau motif ?").
- **Utilisation par des adultes** (ex. : pour des ateliers de programmation créative).

---

4. Synthèse des Impacts Architecturaux

Cas d’Usage	Impact Technique Principal	Complexité
Collaboratif en temps réel	WebSockets + gestion des conflits de code	Élevée
Tortue-robot synchronisée	Communication serveur/MicroPython + calibration	Moyenne
Projection centrale	Séparation frontend/backend + mode lecture seule	Faible
Sauvegarde des projets	Base de données (SQLite) ou fichiers locaux	Faible
Utilisation hors ligne	Service Workers (pour le web) ou app desktop	Moyenne

---

5. Prochaines Étapes pour Toi

1. **Valider les cas d’usage prioritaires** avec les enseignants/élèves :
   - Quels scénarios sont **indispensables** pour la rentrée 2026–2027 ?
   - Quels sont **optionnels** (peuvent attendre une V2) ?
2. **Affiner les contraintes techniques** :
   - Exemple : Si la tortue-robot doit être synchronisée avec la virtuelle, il faudra un **protocole de communication temps réel** (WebSockets + MicroPython).
   - Si le partage de code est collaboratif, il faudra choisir entre **CRDT** (complexe) ou **verrouillage** (plus simple).
3. **Prototypage léger** :
   - Une fois les cas d’usage validés, un **prototype minimal** (ex. : Flask + HTML basique) permettra de tester les flux principaux.

---

6. Exemple de Décisions Architecturales Découlant des Cas

d’Usage

Décision	Si Cas d’Usage X est validé...	Sinon...
Technologie backend	Flask + WebSockets (pour le temps réel)	Flask seul (plus simple)
Stockage des projets	SQLite + système de versions (pour l’historique)	Fichiers locaux (JSON)
Frontend	Vue.js léger (pour la réactivité)	Vanilla JS (plus simple)
Communication tortue-robot	WebSockets + protocole custom	USB sériel (pour un prototype)

Question pour toi : Quels sont les **3 cas d’usage prioritaires** que tu veux absolument voir implémentés pour la rentrée 2026–2027 ? Cela permettra de focaliser l’architecture sur l’essentiel. □□

(Exemple de réponse : "1. Collaboratif en classe avec projection, 2. Tortue-robot synchronisée, 3. Sauvegarde locale des projets.")

1. Principes fondateurs de l’UI (version "puriste mais moderne")

a. Minimalisme radical

- **Pas de blocs prédéfinis** (type Scratch) : L’enfant écrit **toujours** du code texte, mais avec des aides **optionnelles et non intrusives**.
- **Pas de surcharge visuelle** : L’écran est épuré, avec **uniquement** :
  - Une **zone de dessin** (tortue + traces).
  - Une **console de commande** (saisie texte).
  - Une **zone de procédures** (pour sauvegarder/réutiliser du code écrit par l’enfant).

b. Modernité discrète

- **Design sobre mais actuel** :
  - **Couleurs** : Fond clair (blanc cassé ou bleu très pâle) pour éviter la fatigue oculaire, tortue et textes en noir ou couleurs vives **uniquement pour le feedback** (ex. : syntaxe correcte en vert, erreur en orange).
  - **Typographie** : Police monospace (ex. **Courier New** ou **Fira Code**) pour le code (lisibilité et alignement des commandes), et une police simple (ex. **Open Sans**) pour les menus.

- **Formes** : Lignes épurées, pas de boutons superflus. Les éléments interactifs sont **discrets** mais identifiables (ex. : survol = léger soulignement).
- **Feedback visuel intelligent** :
  - La tortue **clignote brièvement** quand une commande est exécutée (sans son, pour éviter la distraction).
  - Les erreurs de syntaxe sont signalées par un **soulignement rouge** + message court en bas de l'écran (ex. : "*Il manque un nombre après AVANCE*").
  - **Pas d'animations inutiles** : La tortue se déplace **immédiatement** (pas de "dance" ou d'effets de style).

c. Personnalisation par l'enfant (sans assistance forcée)

- **Boutons programmables** :
  - L'enfant peut **créer ses propres boutons** en glissant-déposant **du code qu'il a écrit** dans une zone dédiée (ex. : un bouton "CARRÉ" exécutant sa procédure `POUR CARRE ... FIN`).
  - Ces boutons sont **des raccourcis**, pas des commandes prédéfinies. Ils apparaissent comme des **étiquettes textuelles simples** (pas d'icônes imposées).
- **Éditeur de procédures** :
  - Une zone latérale (ou un onglet) permet de **lister les procédures définies par l'enfant**, avec un bouton "Éditer" pour les modifier.
  - **Pas de "bibliothèque" prédéfinie** : Tout doit être écrit par l'enfant ou importé depuis ses propres fichiers.

d. Respect de la philosophie Logo

- **Pas de gamification** (badges, étoiles) : L'enfant est motivé par **la création elle-même**, pas par des récompenses externes.
- **Pas de "mode découverte" guidé** : L'interface ne propose **jamais** de défis ou de tutoriels intrusifs. En revanche, un bouton "?" peut ouvrir un **manuel minimaliste** avec des exemples de code (ex. : "*Pour dessiner un carré : Répète 4 fois [AVANCE 100 TOURNE 90]*").
- **Accès direct au métalangage** :
  - L'enfant peut **voir et modifier** les primitives de base (ex. : redéfinir `AVANCE` s'il le souhaite, comme dans les versions historiques de Logo).

> On est encore en réflexion sur le sujet. J'ai aussi les idées suivante
> - permettre à plusieurs enfants de partager en temps réel des codes ou morceaux de code
> - pouvoir séparer de l'interface la zone d'exécution ou il y a la tortue : elle peut-être projetée par exemple sur un mur, et les enfants travaillent sur un/leur poste.

> - donc ou peut aussi envisager une architecture client/serveur ou un code python/flask sert l'interface ou plusieurs interfaces, dont la zone de projection tortue.
> - enfin, j'inscris ce projet dans un suite de projet ou les enfants peuvent utiliser une tortue-robot réelle, qui se déplace sur un feuille sur le sol de la classe.
> - ce projet sera porté par les enfants ados et la programmation de la tortue pourrait-être faite au travers de la bibliothèque micro-python

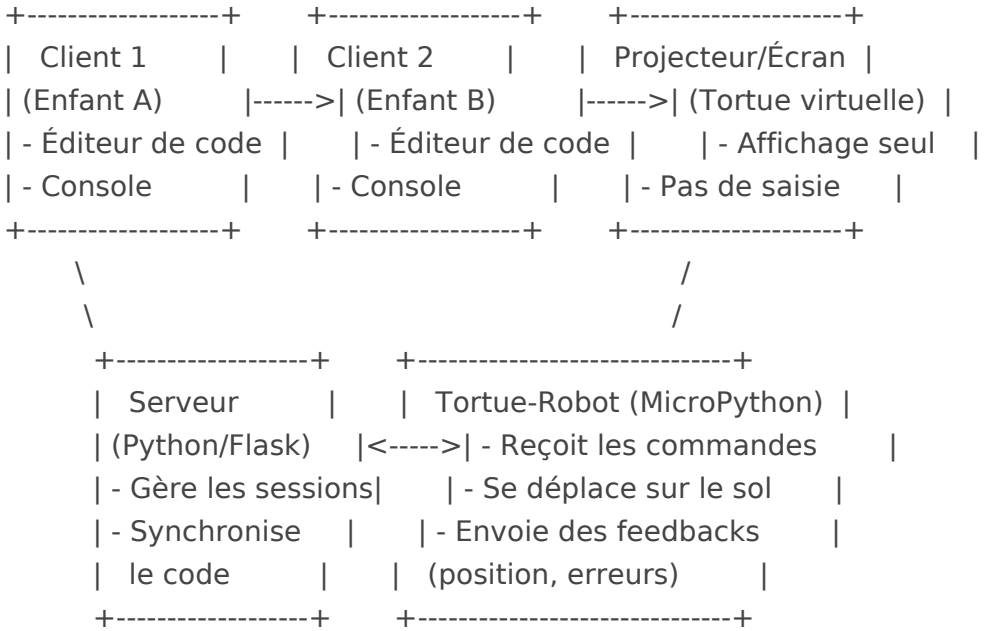
PROPOSITION :

1. Architecture Globale Proposée

Ton projet combine **4 dimensions** :

1. **Interpréteur Logo collaboratif** (partage de code en temps réel).
2. **Séparation interface/exécution** (projection de la tortue sur un mur).
3. **Architecture client/serveur** (pour gérer plusieurs interfaces et la projection).
4. **Intégration avec des tortues-robot physiques** (via MicroPython).

Voici une architecture technique cohérente pour répondre à ces besoins :



Merci pour ces précisions passionnantes ! Ton projet prend une dimension **collaborative, physique et pédagogique** très riche, qui dépasse largement un simple interpréteur Logo classique. Voici une analyse structurée des enjeux et des solutions possibles, en gardant à l'esprit tes contraintes (fidélité à l'esprit Logo, modernité discrète, et intégration avec du matériel comme les tortues-robot).

---

2. Solutions Techniques par Dimension

A. Collaboratif en temps réel

Objectif : Permettre à plusieurs enfants de partager du code ou des morceaux de code. **Solutions** :

- **CRDT (Conflict-free Replicated Data Type)** :
  - Utiliser une bibliothèque comme Yjs (pour le web) ou Automerge (pour une app desktop).
  - Avantages : Synchronisation fluide, pas de conflits, historique des modifications.
  - Exemple : Si un enfant écrit `POUR CARRE`, un autre peut voir la définition en temps réel et l'utiliser.
- **WebSockets (pour une solution client/serveur)** :
  - Le serveur Flask diffuse les mises à jour de code à tous les clients connectés.
  - Exemple avec Flask-SocketIO :
    ```
    from flask_socketio import SocketIO, emit

    @socketio.on('code_update')
    def handle_code_update(data):
        emit('code_update', data, broadcast=True)  # Envoie à tous les clients
    ```
- **Alternative simple** :
  - Un système de "copier-coller partagé" via le serveur (moins fluide, mais plus simple à implémenter).

B. Séparation interface/exécution (projection)

Objectif : La tortue s'affiche sur un écran/mur, tandis que les enfants codent sur leurs postes.
Solutions :

- **Mode "Projection"** :
  - Le serveur Flask génère une **URL dédiée** pour la projection (ex. : `/projection`).
  - Cette page n'a **que la zone de dessin** (tortue + traces), sans éditeur ni console.
  - Les enfants envoient leurs commandes au serveur, qui les exécute et met à jour l'affichage pour tous.
- **Technos** :
  - **Frontend** : HTML/Canvas pour la projection (même code que les clients, mais en mode "lecture seule").
  - **Backend** : Flask gère les états partagés (position de la tortue, couleurs, etc.).
- **Exemple de flux** :
  1. Un enfant tape `AVANCE 50` sur son poste.
  2. Le code est envoyé au serveur via WebSocket.
  3. Le serveur met à jour l'état de la tortue et envoie la nouvelle position à **tous les clients + la projection**.

C. Architecture Client/Serveur (Flask)

Objectif : Centraliser la logique pour gérer plusieurs interfaces et la projection. **Solutions** :

- **Flask + SocketIO** :
  - **Avantages** :
    - Léger et facile à déployer (même sur un Raspberry Pi en classe).
    - Permet de gérer les sessions, l'historique des commandes, et la synchronisation.
  - **Exemple de routes** :
    ```
    @app.route('/')
    def index():
        return render_template('editeur.html')  # Interface complète pour les enfants

    @app.route('/projection')
    def projection():
        return render_template('projection.html')  # Zone de dessin seule
    ```
- **Stockage des projets** :
  - Utiliser une base **SQLite** (légère, sans serveur) pour sauvegarder les procédures des enfants.
  - Exemple de table :
    ```
    CREATE TABLE projets (
        id INTEGER PRIMARY KEY,
        nom TEXT,
        code TEXT,
        date_modification DATETIME
    );
    ```

D. Intégration avec les tortues-robot (MicroPython)

Objectif : Permettre aux enfants de contrôler une tortue physique via le même code Logo.
Solutions :

- **Protocole de communication** :
  - Le serveur Flask agit comme un **pont** entre les clients et la tortue-robot.
  - Les commandes Logo sont traduites en instructions MicroPython (ex. : `AVANCE 50` → `robot.forward(50)`).
- **Matériel** :
  - La tortue-robot est équipée d'un **ESP32 ou Raspberry Pi Pico** (compatible MicroPython).
  - Elle se connecte au serveur via **Wi-Fi** (pour les ESP32) ou **USB** (pour un prototype simple).
- **Exemple de code MicroPython** (côté robot) :

```
from machine import Pin, PWM
import network
import socket

# Configuration Wi-Fi
sta_if = network.WLAN(network.STA_IF)
sta_if.active(True)
sta_if.connect('SSID', 'password')

# Moteurs (exemple simplifié)
def forward(steps):
    # Logique pour avancer
    print(f"Avance de {steps} pas")

# Serveur TCP pour recevoir les commandes
s = socket.socket()
s.bind(('0.0.0.0', 8080))
s.listen(1)

while True:
    conn, addr = s.accept()
    data = conn.recv(1024).decode()
    if data.startswith("AVANCE"):
        steps = int(data.split()[1])
        forward(steps)
    conn.close()
```

- **Côté serveur Flask** :
  - Quand un enfant envoie `AVANCE 50`, le serveur transmet la commande à la tortue-robot via Wi-Fi/USB.

---

3. Points d’Attention

- **Latence** :
  - Avec plusieurs enfants + une tortue physique, la latence peut devenir visible.
  - **Solution** : Utiliser des WebSockets pour minimiser les délais, et prévoir un système de "file d’attente" pour les commandes.
- **Sécurité** :
  - Si le serveur est accessible sur le réseau de l’école, sécuriser les connexions (mot de passe simple pour la classe).
- **Pédagogie** :
  - **Ne pas cacher la complexité** : Les enfants doivent comprendre que la tortue physique et la tortue virtuelle suivent les mêmes règles.
  - **Feedback clair** : Si la tortue-robot bute sur un obstacle, envoyer un message comme "*Attention, la tortue est bloquée !*" sur tous les écrans.

---

4. Exemple de Workflow Complet

1. **En classe** :
   - Les enfants ouvrent leur navigateur sur `http://<adresse-du-serveur>`.
   - Ils voient l’éditeur Logo et la console.
   - Le projecteur affiche `http://<adresse-du-serveur>/projection`.
2. **Collaboration** :
   - Alice écrit `POUR CARRE ... FIN`.
   - Bob peut voir la procédure et l’utiliser dans son propre code.
3. **Exécution** :
   - Quand un enfant exécute `CARRE`, la tortue virtuelle (projection) et la tortue-robot (si connectée) dessinent un carré.
4. **Feedback** :
   - Si la tortue-robot rencontre un problème, un message s’affiche sur tous les écrans.

---

5. Technologies Recommandées

Composant	Technologie	Pourquoi
Frontend	HTML/CSS/JS (Vanilla ou Vue.js léger)	Simple, portable, pas de compilation.
Backend	Python/Flask + Flask-SocketIO	Léger, facile à déployer, compatible avec MicroPython.
Base de données	SQLite	Sans serveur, idéale pour un usage en classe.
Tortue-robot	ESP32 + MicroPython	Peu coûteux, programmable en Python, Wi-Fi intégré.
Collaboration	WebSockets (SocketIO)	Temps réel, simple à implémenter avec Flask.

---

6. Prochaines Étapes Possibles

1. **Prototype minimal** :
   - Un serveur Flask basique + une page HTML avec un éditeur et un canvas.
   - Test avec 2 clients et une projection.
2. **Intégration MicroPython** :
   - Connecter une tortue-robot (ESP32) au serveur et traduire les commandes Logo.
3. **Tests en classe** :
   - Observer comment les enfants interagissent avec l'outil (collaboration, compréhension des feedbacks).

Question pour toi :

- Veux-tu que je te propose un **exemple de code Flask + HTML/JS** pour démarrer le prototype (avec la synchronisation basique et la projection) ?
- Ou préfères-tu approfondir un autre aspect (ex. : la communication avec la tortue-robot, ou la gestion des conflits de code) ?