Rapport de projet :
Application de Planning Poker

Master 1 Informatique — Conception Agile de Projets Informatiques

Auteurs : AMY Léo & LOREL Guillaume

Date de rendu : 19 Décembre 2025

Sommaire

1. Introduction

1.1. Contexte du projet et enjeux

Ce projet a été réalisé dans le cadre du cours "Conception Agile de Projets Informatiques", et porte sur la conception d'une application de Planning Poker. Outil central des méthodologies agiles telles que Scrum ou l'eXtreme Programming, le Planning Poker permet aux équipes d'estimer collectivement la complexité des tâches d'un projet. L'objectif principal de notre application est de dématérialiser ce processus, historiquement réalisé avec des cartes physiques, afin de le rendre plus fluide et adapté aux environnements numériques. L'application a été pensée pour un usage local, facilitant le vote successif de chaque membre de l'équipe sur un poste unique ou projeté.

1.2. Objectifs pédagogiques

Au-delà de l'aspect fonctionnel, ce projet avait pour vocation de valider des compétences techniques et méthodologiques précises. Il s'agissait notamment de démontrer notre capacité à structurer une architecture logicielle (MVC dans notre cas), tout en conservant une approche agile (pair programming, intégration continue, etc.). La collaboration en binôme et la gestion des données pour assurer la robustesse de l'application constituaient également des axes importants du projet.

1.3. Organisation du binôme

Nous avons choisi de réaliser l'intégralité du développement en Pair Programming, un choix motivé par la complémentarité de nos profils. En effet, notre binôme a été constitué dans une optique d'**échange de compétences** : Léo apportait une aisance technique plus marquée sur le langage Python, la logique algorithmique et la structure MVC, tandis que Guillaume disposait d'une affinité plus forte pour le développement Frontend et l'ergonomie des interfaces.

Contrairement à une organisation classique où chacun se serait cantonné à son domaine de prédilection (Backend vs Frontend), nous avons travaillé conjointement sur la même PC. Cette méthode a renforcé nos compétences : le développement et la réflexion des interfaces a permis à Léo de monter en compétence sur la **gestion graphique** et l'**expérience utilisateur**, tandis que l'implémentation du moteur de jeu a renforcé la **maîtrise Python** de Guillaume.

Cette organisation a également eu un impact direct sur la qualité du projet. La revue de code étant continue, chaque ligne a été validée instantanément par nous deux, réduisant les erreurs. Enfin, les choix architecturaux importants ont pu être décidés immédiatement et avec une vision globale grâce à cette proximité.

2. Analyse et choix techniques

2.1. Environnement de développement en Python

Le choix du langage a fait l'objet d'une vraie discussion au lancement du projet. On a longuement hésité à partir sur une application web, ce qui est souvent la norme pour ce genre d'outil. Mais nous avons finalement décidé d'utiliser Python pour développer une application de bureau.

La raison est double : d'une part, Python permet de **prototyper très vite**, ce qui est crucial sur un projet court. D'autre part, c'était une volonté commune de monter en compétence sur ce langage, notamment au vu de son importance dans le Master en général.

2.2. Interface graphique : de Qt Designer à CustomTkinter

Le choix de la technologie d'interface utilisateur a été difficile à faire au début du projet. Notre première intention était d'utiliser le **framework Qt (via PySide6)**, réputé pour sa robustesse et son historique. Nous avons commencé par maquetter nos vues à l'aide de l'outil graphique **Qt Designer**, pensant gagner du temps grâce au "Drag & Drop".

Cependant, nous nous sommes rapidement heurtés à deux limites bloquantes. D'une part, l'inadéquation esthétique : le style natif des widgets Qt, très "industriel" et ancien, ne correspondait pas à l'identité visuelle moderne que nous souhaitions, et la personnalisation demandait une lourde surcharge de travail en feuilles de style CSS (QSS). D'autre part, la lourdeur du workflow : le cycle "Design dans l'outil -> Génération du fichier .ui -> Conversion en Python -> Intégration dans le contrôleur" s'est révélé trop rigide, freinant notre développement.

Nous nous sommes donc tournés vers **CustomTkinter**, une bibliothèque basée sur le toolkit standard de **Python (tkinter)**. Ce choix s'est révélé être payant : l'approche "Code-First" (interface définie entièrement en Python) rend le code plus facile à versionner et à modifier dynamiquement. De plus, les composants offrent nativement un **rendu moderne** (boutons arrondis) et un **mode sombre** sans configuration complexe.

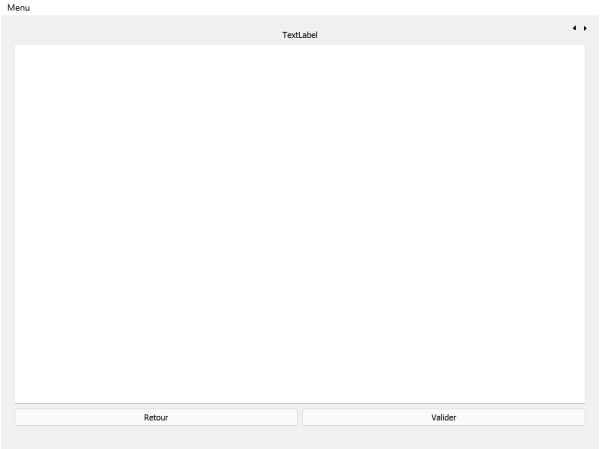

Figure 1 : *Maquette initiale réalisée sous Qt Designer. L'interface apparaît basique et le style "natif" ne correspondait pas à l'identité visuelle souhaitée.*

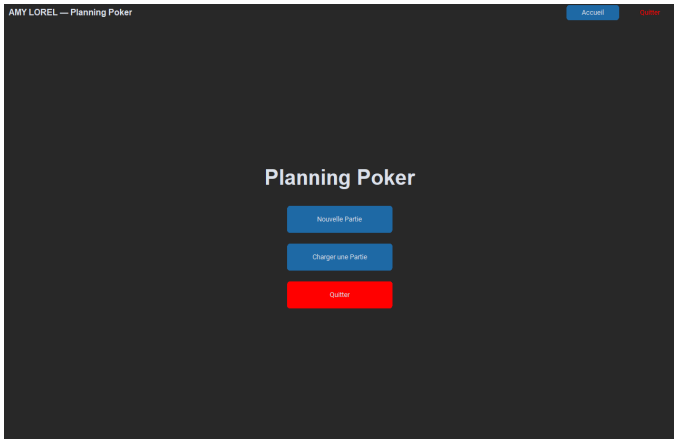

Figure 2 : *Interface finale développée avec la librairie CustomTkinter. Le rendu est plus moderne, épuré et intègre nativement un thème sombre.*

2.3. Architecture logicielle : application en MVC

Pour garantir la maintenabilité, nous avons mis en place une architecture Modèle-Vue-Contrôleur (MVC) qui sépare clairement les responsabilités :

La couche des **Modèles** (models/) représente la logique métier pure. La classe GameSession est le pivot du projet : elle détient l'état complet de la partie. Elle est assistée par GameRules (qui encapsule les algorithmes de moyenne/médiane) et par les entités Player et Backlog.

Les **Vues** (views/) sont passives et héritent de ctk.CTkFrame. Elles ne prennent aucune décision métier. Elles exposent des méthodes de mise à jour (ex: refresh_ui()) et capturent les événements (clics) pour les déléguer immédiatement au contrôleur via des callbacks.

Enfin, les **Contrôleurs** (controllers/) agissent comme le cerveau de l'application. Le MainController gère la navigation, tandis que le GameController gère la mécanique du jeu : il vérifie la fin du tour, applique les règles de calcul via le modèle, et décide de la transition vers l'écran de résultat ou de revote.

2.4. Modélisation des données (UML)

La structure de nos données repose sur une architecture centralisée. Comme le montre le diagramme de classes ci-dessous, nous avons conçu le modèle autour de la classe **GameSession**, qui agit comme le conteneur unique de l'état de l'application.

Cette classe "maîtresse" rassemble les différentes entités du domaine :

- **Backlog :** Encapsule la liste des User Stories à estimer.
- **GameRules :** Gère la configuration de la partie (le mode de validation choisi, comme l'Unanimité ou la Majorité).

- **Player :** Représente les participants identifiés par leur nom.

Au-delà de ces relations structurelles, GameSession a la responsabilité de suivre la dynamique du jeu. C'est elle qui maintient les variables d'état telles que :

- current_feature_index : Un curseur pointant vers la tâche en cours de vote.
- current_round_number : Le numéro du tour actuel (nécessaire pour gérer les revotes).
- votes et validated_features : Des dictionnaires stockant respectivement les votes temporaires du tour et les résultats finaux validés.

Cette modélisation simplifie considérablement la gestion de l'application : n'importe quelle vue ou contrôleur ayant accès à l'instance unique de GameSession dispose immédiatement de tout le **contexte nécessaire** (qui joue ? quelle règle ? où en est-on ?).

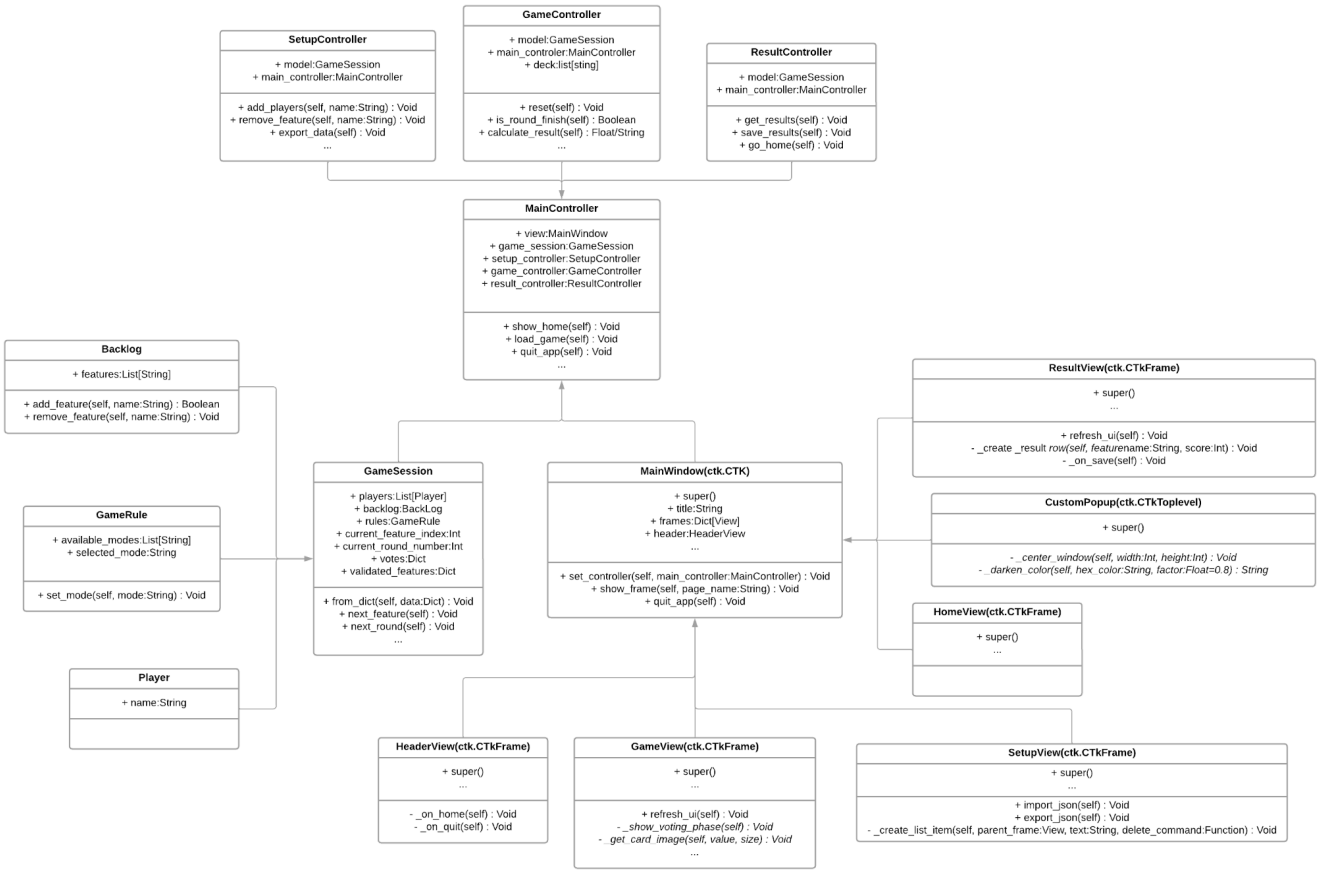

***Figure 3** : Diagramme de classes UML de l'application. On y distingue l'architecture MVC avec le MainController supervisant les contrôleurs spécifiques (Setup, Game, Result) et la classe GameSession centralisant les données du modèle (Joueurs, Backlog, Règles).*

2.5. Sérialisation d'états en JSON

La règle "Pause Café" imposait de pouvoir interrompre et reprendre une partie sans perte. Cela impliquait de ne pas simplement sauvegarder des résultats finaux, mais de sérialiser l'état complet de la machine à états.

Nous avons implémenté un système de sauvegarde au format JSON. Dans la classe GameSession, la méthode to_dict() capture un instantané précis comprenant la **configuration** de la partie (liste des joueurs, règles choisies), l'**état d'avancement** (index de la User Story en cours dans le backlog) ainsi que l'**état du tour**. L'**historique** des tâches déjà validées est également conservé.

Figure 4 : *Extrait d'un fichier de sauvegarde. Il contient l'état complet de la partie : statut, règles, liste des joueurs, backlog et index de la tâche en cours, permettant une reprise exacte du jeu.*

Au chargement (from_dict), le contrôleur restaure cet état et redirige l'utilisateur : soit vers la table de jeu si la partie était en cours, soit vers le bilan si elle était terminée.

3. Méthodologie de développement

3.1. Approche agile centrée sur le Pair Programming

Pour répondre à l'une des consignes principales de ce projet, nous avons placé le *Pair Programming* au cœur de notre méthodologie. Afin d'appliquer ce principe, nous avons privilégié le travail en présentiel, limitant au maximum le développement à distance. Cette proximité nous a permis de suivre systématiquement le même cycle : une phase de conception orale pour discuter des choix d'implémentation, immédiatement suivie d'une phase de codage à “quatre mains”.

Cette approche, où un pilote écrit sous la supervision constante du copilote, a non seulement favorisé la communication directe mais a également quasiment éliminé les bugs de logique souvent rencontrés lors de l'intégration de composants, développés séparément.

3.2. Gestion de versions et transition vers l'intégration continue (CI)

Concernant la gestion des versions, nous avons adopté un workflow organisé. La totalité du développement s'est effectuée sur une branche commune develop, permettant à la branche main d'accueillir uniquement les **versions stables**. Initialement manuel, notre workflow a évolué vers une **automatisation complète**.

Au départ, nous avons créé deux fichiers de configuration distincts : un pour lancer nos tests et un autre pour générer la documentation. Nous avons vite réalisé que cette séparation dispersait l'information et rendait la maintenance pénible. Nous avons donc fait le choix de tout regrouper dans un unique pipeline d'Intégration continue (CI) via GitHub Actions. Désormais centralisé dans le fichier .github/workflows/actions.yaml, il est configuré pour agir selon deux scénarios : le développement courant (sur chaque push) et la publication officielle (via des tags).

Cette transition nous a permis d'automatiser trois aspects :

- **Tests Unitaires (python -m unittest) :** Les tests sont lancés automatiquement à chaque modification du code. Nous avons ciblé la logique métier pour garantir la fiabilité des règles du jeu :
    - *Règles de calcul :* Validation des algorithmes de moyenne et de médiane (arrondis, gestion des valeurs non numériques comme le "?").
    - *Mécanique de vote :* Vérification de la détection de l'unanimité.
    - *Intégrité des données :* Tests sur l'ajout et la validation des User Stories dans le Backlog.

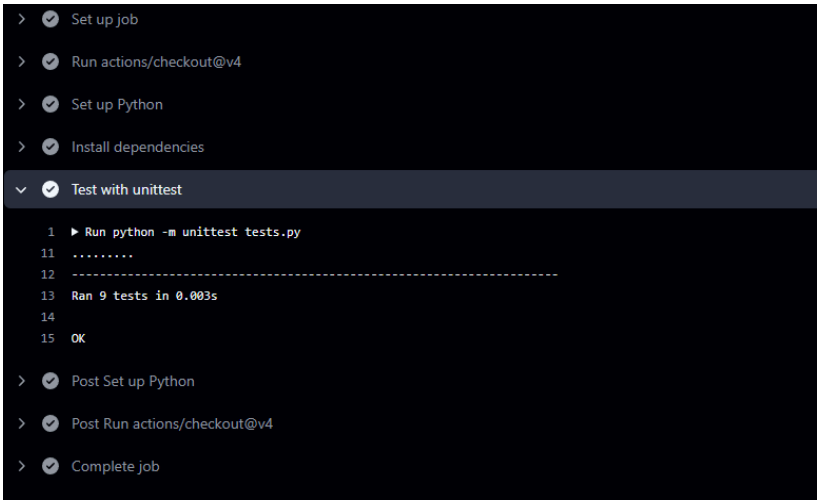

Figure 5 : *Capture d'écran d'un build de l'application lors d'un push avec l'exécution des tests.*

- **Génération de Documentation :** Nous utilisons l'outil **Doxygen**. Le fichier Doxyfile contrôle l'extraction des commentaires du code. Un job spécifique dans la CI génère la documentation au format HTML et la rend accessible directement via les **GitHub Pages**. Cela permet de consulter une documentation technique toujours à jour via une simple URL Web, sans avoir à parcourir le code ou cloner le projet.

Figure 6 : *Capture d'écran de la pipeline pages-build-deployment qui déploie la doc dans Github Pages, permettant d'y accéder depuis un url*

- **Création d'un exécutable (.exe) :** Pour la version 2, nous avons voulu rendre le jeu accessible sans imposer l'installation de Python. Nous avons ajouté une étape qui utilise **PyInstaller** pour compiler le projet en un fichier exécutable unique pour Windows. Cette construction se déclenche automatiquement lors de la publication d'une "Release", fournissant aux utilisateurs un fichier .exe clé en main directement dans les artefacts GitHub.

3.3. Usage de l'IA

Dans une démarche moderne de développement, nous avons ponctuellement utilisé des assistants basés sur l'IA, notamment **Gemini** et **GitHub Copilot**. Il est important de préciser que ces outils n'ont pas remplacé notre réflexion : ils ont agi comme des assistants techniques. Nous les avons principalement sollicités pour trois types de tâches : l'analyse de messages d'erreurs (debugging), la proposition de structures pour la gestion des exceptions (blocs try/except), et la génération de squelettes de code pour les tests unitaires. Cette assistance nous a permis de gagner un temps précieux sur les tâches répétitives pour mieux nous concentrer sur l'architecture du projet et la logique métier.

4. Manuel utilisateur et fonctionnalités

4.1. Installation et déploiement

Pré-requis techniques : Python 3.10 ou ultérieur. Procédure :

1. Récupération du code source.
   git clone https://github.com/LeoAugustinAmy/AMY_LOREL_PlaningPoker.git
   cd AMY_LOREL_PlaningPoker.git

2. Installation des dépendances : pip install -r requirements.txt.
3. Lancement : python main.py.

4.2. Parcours utilisateur : initialisation et configuration

L'écran d'accueil permet de démarrer une session ou de charger une sauvegarde. Ensuite, l'écran de **configuration (setup)** permet au Scrum Master de définir plusieurs éléments. Il est divisé en trois zones clés : la zone joueurs (ajout dynamique), la zone backlog (saisie des User Stories), et la zone règles.

Un point important est la sélection du mode de calcul pour les tours : Moyenne, Médiane ou Majorité. Conformément au cahier des charges, nous avons ajouté une fonctionnalité d'import/export permettant de charger une configuration (joueurs et tâches) depuis un JSON existant.

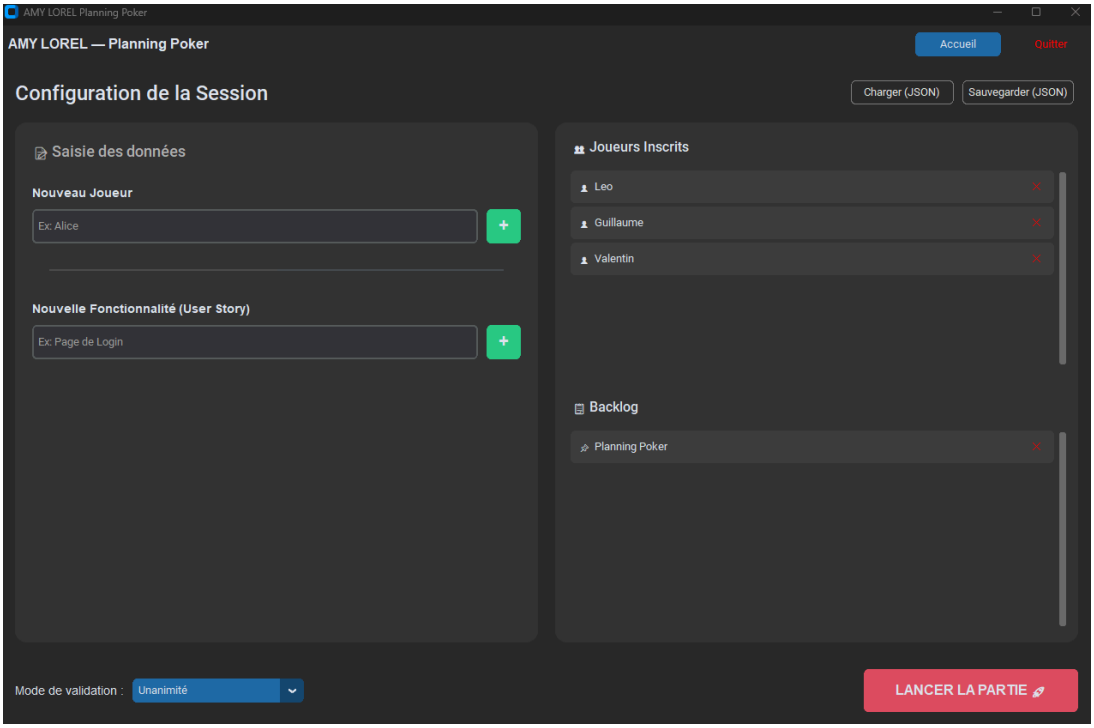

Figure 7 : *Écran de configuration (Setup). Le Scrum Master peut y ajouter dynamiquement des joueurs, saisir les User Stories du Backlog et définir la règle de validation (Unanimité, Majorité, etc.) avant de lancer la partie.*

4.3. Phase de jeu : mécanismes de vote et règles

L'interface simule une table de poker. L'application indique qui a la main ("C'est à Alice de voter"). Le joueur sélectionne une carte, qui apparaît cachée pour ne pas influencer les autres.

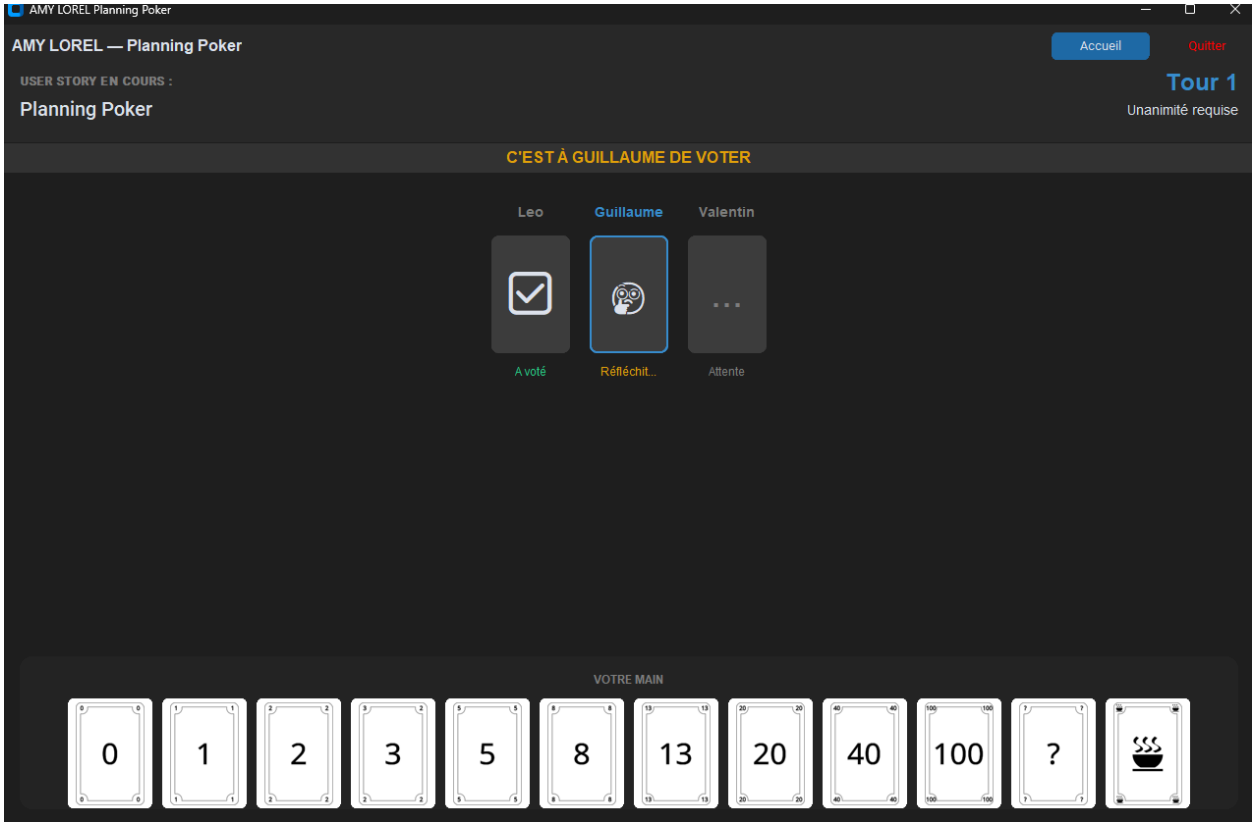

Figure 8 : *Interface principale de jeu (Phase de vote). On y voit la User Story en cours de traitement, le statut de chaque joueur (A voté, Réfléchit...) et la main de cartes disponible en bas de l'écran.*

L'application gère strictement les règles du sujet. Lors du tour 1, le système impose l'unanimité parfaite ; si les valeurs diffèrent, une alerte "Pas de consensus" force le débat. Pour les tours suivants, si le consensus n'est pas atteint, la règle choisie en configuration (ex: Moyenne) s'applique automatiquement pour trancher.

4.4. Fonctionnalité "Pause Café"

Si l'ensemble des joueurs sélectionne la carte "☕", l'application détecte une volonté d'interruption. Elle déclenche alors automatiquement la sauvegarde de l'état complet dans un fichier JSON et le retour au menu principal. Cette fonctionnalité assure qu'aucune estimation validée n'est perdue.

5. Bilan et retour d'expérience

5.1. Bilan technique

Ce projet a validé la pertinence de l'architecture MVC. Bien que générant plus de fichiers, cette séparation a été cruciale lors de l'ajout de la fonctionnalité de sauvegarde : nous avons pu implémenter cette mécanique complexe en modifiant uniquement le Modèle et le Contrôleur, sans impacter une seule ligne des Vues. Le pivot de Qt vers CustomTkinter nous a également appris l'importance de choisir des **outils adaptés** à l'échelle du projet plutôt que de s'obstiner sur des standards parfois trop lourds pour le besoin réel.

Enfin, nous avons beaucoup appris sur l'aspect “DevOps”. La mise en place du pipeline CI/CD via GitHub Actions nous a montré qu'un projet ne s'arrête pas à l'écriture du code. L'automatisation complète de l’application, des tests unitaires jusqu'à la génération de l'exécutable final (.exe) pour Windows a permis de transformer notre dépôt en un logiciel potentiellement distribuable.

5.2. Bilan méthodologique

Ce projet nous a permis de mettre en pratique de nombreuses compétences techniques, mais c'est surtout sur la gestion de projet et la méthodologie que nous avons le plus appris.

Nous avons commencé le développement en pratiquant beaucoup le **pair programming**. Au début, c'était très efficace pour niveler nos connaissances et s'assurer que nous comprenions tous les deux la structure du code. Cependant, nous nous sommes rendu compte que cette méthode a ses limites, surtout sur un projet plus conséquent comme celui-ci. Quand les délais sont devenus courts en fin de projet, travailler constamment à deux sur le même clavier nous a parfois ralentis.

Avec le recul, nous réalisons aussi que nous aurions dû adopter une approche plus **itérative**. Nous avons eu tendance à vouloir développer de gros blocs de fonctionnalités d'un coup, ce qui a rendu l'intégration finale plus complexe. Travailler par petits cycles courts nous aurait permis de valider le fonctionnement étape par étape.

Bien que notre pipeline soit aujourd'hui complet et performant (tests, documentation, build), nous l'avons mis en place tardivement. Si c'était à refaire, nous implémenterions la CI dès le début. Cela nous aurait épargné de détecter manuellement des régressions que les tests automatiques auraient pu signaler directement.