université PARIS-SACLAY
IUT D'ORSAY

FFB
FEDERATION FRANÇAISE DE BRIDGE

STAGE DE FIN D'ÉTUDES

RAPPORT D'ACTIVITÉ

SmartKibitz

Système de supervision intelligente
de tournois de bridge par vision artificielle

PRÉSENTÉ PAR

Youssef Abichou

Étudiant en BUT3 Informatique

SOUS LA DIRECTION DE

Zakaria Oulalite (Maître de stage)

Marie-Paule Behar (Tutrice)

Fédération Française de Bridge ▪ 3 février – 15 mai 2026

Remerciements

Ce stage n'aurait tout simplement pas existé sans quelques personnes clés.

Hervé Borredon, enseignant à l'IUT d'Orsay, m'a enseigné Django et a été le tuteur de mon projet universitaire BridgeVision [1]. C'est la qualité de ce projet, et la confiance qu'il m'a accordée tout au long de son développement, qui ont rendu cette opportunité possible. Il a constitué le premier lien entre mon travail académique et la Fédération Française de Bridge.

Laurent Danziger, chef du département informatique de la FFB, a fait le choix de m'accueillir en stage après avoir découvert BridgeVision. C'est lui qui a vu dans ce projet universitaire le potentiel d'une vraie mission à la Fédération. Je lui dois la liberté de conception et les équipements sur lesquels j'ai pu travailler. Laurent est aussi un excellent bridgeur — ce qui n'a pas manqué d'enrichir nos échanges sur les besoins réels du terrain.

Marie-Paule Behar, ma tutrice pédagogique à l'IUT d'Orsay, a joué un rôle essentiel dans l'aboutissement de ce stage. Son suivi rigoureux, ses conseils sur la conduite du projet et son engagement dans la relation entre l'IUT et la FFB ont été déterminants pour que cette opportunité se concrétise. Sa contribution dépasse largement le cadre formel du suivi pédagogique.

Zakaria Oulalite, mon maître de stage, a assuré un encadrement direct, fluide et bienveillant. Ses encouragements et ses conseils pour la suite m'ont été précieux.

Je remercie **Vincent Lamaire**, référent technique de la Fédération et bridgeur, pour les pauses déjeuner enrichissantes et les codes professionnels qu'il m'a aidé à déchiffrer.

Je remercie enfin l'ensemble de l'équipe de la **Fédération Française de Bridge** pour l'accueil dans une institution à l'histoire et à la culture singulières.

Youssef Abichou
Avril 2026

1. Démo interactive disponible sur mon portfolio : `https://youssef.cc/projects/bridgevision`

Résumé

Ce rapport présente le travail réalisé lors d'un stage de fin d'études de BUT3 Informatique à la Fédération Française de Bridge (FFB), du 3 février au 15 mai 2026. La mission consiste à concevoir et développer **SmartKibitz**, un système de supervision intelligente de tournois de bridge par vision artificielle distribuée.

Le bridge en compétition mobilise plusieurs tables simultanées sur lesquelles des opérateurs doivent saisir manuellement chaque carte jouée dans le logiciel de scoring. SmartKibitz vise à automatiser cette saisie par reconnaissance visuelle, tout en offrant à l'arbitre une vue globale et un outil de relecture vidéo. Le système est dimensionné pour 20 tables — capacité définie avec la FFB pour couvrir l'intégralité d'une salle de tournoi. La contrainte principale est de construire un système *air-gapped* — sans accès à Internet — capable de fonctionner avec sa propre infrastructure réseau.

La solution déployée repose sur un Raspberry Pi 5 par table, équipé d'un accélérateur neuronal Hailo AI HAT (26 TOPS) et de deux caméras Sony IMX708. Un modèle YOLOv8m, initialement entraîné dans le cadre du projet universitaire BridgeVision puis raffiné sur des données réelles collectées à la FFB, détecte les 52 types de cartes en temps réel avec une latence d'environ 50 ms. Un serveur central Django 5.2 (ASGI, Django Channels, Redis) agrège les détections de toutes les tables via WebSocket et les restitue sur un tableau de bord interactif accessible depuis n'importe quel navigateur du réseau local.

Le projet couvre également le provisionnement automatique des Raspberry Pi (*Zero-Touch Provisioning* par scan ARP et déploiement SSH/SCP) et la mise à jour à distance du code des nœuds edge par échange de symlink atomique (*OTA Hot-Swap*).

Au terme du stage, deux tables sont opérationnelles. L'infrastructure est prête à accueillir les dix-huit tables restantes.

Mots-clés : vision par ordinateur, IA embarquée, WebSocket, Raspberry Pi, Hailo NPU, YOLOv8, Django ASGI, Zero-Touch Provisioning, event sourcing, air-gap.

Abstract

This report presents the work carried out during a final-year internship (French *BUT3 Informatique*) at the Fédération Française de Bridge (FFB), from February 3 to May 15, 2026. The mission was to design and develop **SmartKibitz**, a distributed AI-based real-time supervision system for bridge tournaments.

Competitive bridge involves up to 20 simultaneous tables requiring active arbitration to detect rule violations. The primary constraint was to build a fully air-gapped system — with no internet access — operating on a dedicated local network inside the tournament venue.

The deployed solution places a Raspberry Pi 5 at each table, fitted with a Hailo AI HAT neural accelerator (26 TOPS) and two Sony IMX708 camera modules. A YOLOv8m model, initially trained as part of an academic project and subsequently fine-tuned on real FFB match footage, detects all 52 playing card types in real time at approximately 50 ms latency. A central server running Django 5.2 (ASGI, Django Channels, Redis) collects card detection events from all tables over persistent WebSocket connections and delivers them to an interactive dashboard accessible from any browser on the local network.

The project also covers automated provisioning of the Raspberry Pi units (Zero-Touch Provisioning via ARP scanning and SSH/SCP deployment) and remote over-the-air code updates to edge nodes using atomic symlink swaps (OTA Hot-Swap).

At the end of the internship, two tables are fully operational. The infrastructure is ready to be extended to the remaining eighteen tables.

Keywords : computer vision, edge AI, WebSocket, Raspberry Pi, Hailo NPU, YOLOv8, Django ASGI, Zero-Touch Provisioning, event sourcing, air-gap.

Table des matières

Chapitre 1

Contexte

1.1 Une scène de tournoi

Il est 14h30. Dans la salle de jeu du siège de la Fédération Française de Bridge, vingt tables sont occupées. Quatre-vingts joueurs disposent leurs cartes, les retournent, les posent. Un arbitre se lève, traverse la salle en diagonale pour inspecter une contestation à la table 7, revient, reçoit immédiatement un signal à la table 12. Pendant ce temps, à la table 3, une carte vient peut-être d'être jouée en dehors du tour. Personne ne l'a vu.

Ce n'est pas une situation exceptionnelle. C'est le quotidien de tout arbitre de tournoi de bridge.

Le bridge est un jeu de cartes à quatre joueurs, structuré en paires (Nord/Sud contre Est/Ouest), qui se dispute en compétition sous forme de tournois dits *duplicate* : les mêmes distributions de cartes sont jouées sur plusieurs tables simultanément, ce qui permet de comparer les performances indépendamment de la chance. Un tournoi classique mobilise entre 10 et 20 tables, parfois davantage. Les arbitres sont responsables de l'intégrité du jeu — vérifier qu'aucune règle n'est enfreinte, trancher les litiges, détecter les *renonces* (cartes jouées incorrectement). Mais un arbitre humain ne peut pas être à vingt endroits à la fois.

C'est le problème que ce stage m'a demandé de résoudre.

1.2 La mission : automatiser la surveillance

La Fédération Française de Bridge (FFB), fondée en 1933, est l'organisme national qui regroupe plus de 100 000 licenciés et organise les compétitions officielles sur tout le territoire. Sa direction informatique souhaitait explorer une approche technologique pour assister les arbitres : placer des caméras sur chaque table, identifier les cartes posées en temps réel par intelligence artificielle, et centraliser l'information sur un tableau de bord unique consultable par l'arbitre depuis n'importe quel point de la salle.

La contrainte fondamentale était d'ordre pratique : le système devrait fonctionner dans une salle de tournoi sans connexion à Internet, sur un réseau local dédié déployé pour l'occasion. Pas de cloud, pas de services externes. Un réseau isolé, une infrastructure maîtrisée de bout en bout.

J'ai rejoint la FFB le 3 février 2026 pour concevoir et développer ce système, baptisé

SmartKibitz, en solo, avec une liberté de conception totale sur les choix d'architecture, de matériel et d'implémentation.

1.3 Ce que j'ai construit

SmartKibitz est un système distribué de supervision de tournois de bridge par vision artificielle. Chaque table de jeu est équipée d'un mini-ordinateur (Raspberry Pi 5) sur lequel tourne un modèle d'intelligence artificielle capable de reconnaître les 52 types de cartes d'un jeu standard. L'information détectée remonte en temps réel via le réseau local vers un serveur central, visible sur un tableau de bord accessible depuis n'importe quel navigateur.

Le projet repose sur plusieurs briques techniques interdépendantes :

— un **pipeline de vision par ordinateur** : deux caméras par table, fusion d'images, inférence sur accélérateur neuronal dédié ;
— un **serveur temps réel** : Django 5.2 en mode ASGI avec Django Channels et Redis, communication WebSocket bidirectionnelle avec 20 nœuds simultanés ;
— un **tableau de bord interactif** : grille 5 × 4 avec aperçu vidéo de chaque table, statistiques système, historique des cartes jouées, alertes ;
— un **système de déploiement automatisé** : les Raspberry Pi sont provisionnés et mis à jour sans intervention manuelle sur le matériel (*Zero-Touch Provisioning* et *OTA Hot-Swap*).

1.4 Ce que ce stage m'a apporté

Je ne m'attendais pas à travailler sur de l'architecture réseau, du design 3D de boîtier, ou un plan de déploiement matériel à l'échelle. J'avais candidaté pour un stage de développement logiciel. Ce que j'ai réalisé en trois mois déborde largement de cette définition.

Ce stage m'a confronté à des problèmes que l'IUT ne simule pas : un réseau physique qui tombe parce qu'on a saturé la bande passante, un serveur DHCP personnel qui coupe accidentellement l'accès à Internet de toute la Fédération, un capteur qui ne répond plus après une mise à jour système et pour lequel il faut trouver un plan B en quelques heures. Ces situations ont quelque chose d'irremplaçable : elles sont réelles, elles ont des conséquences réelles, et les solutions que l'on trouve ont un impact réel.

J'ai aussi découvert ce que signifie travailler dans une institution qui a une histoire — la FFB n'est pas une startup, elle a 90 ans — et ce que cela implique en termes de rythme, de communication et de décision collective.

1.5 Plan du rapport

Le rapport s'articule en cinq parties. La **Partie 1** présente le contexte du stage : la Fédération Française de Bridge, le bridge en compétition, et les contraintes spécifiques de la mission. La **Partie 2** couvre l'analyse du besoin et les choix de conception. La **Partie 3**, la plus dense, détaille l'implémentation du serveur, des nœuds edge, du provisionnement et du déploiement OTA. La **Partie 4** revient sur les tests, les bugs significatifs et ce qu'ils m'ont appris. La **Partie 5** dresse le bilan des résultats obtenus et des perspectives pour la suite du projet. La **Conclusion** synthétise les apports professionnels et personnels de ce stage.

Chapitre 2

Contexte du stage

2.1 La Fédération Française de Bridge

2.1.1 Une institution centenaire

La Fédération Française de Bridge (FFB) est l'organisme officiel qui régit la pratique compétitive du bridge en France. Fondée en 1933, elle regroupe aujourd'hui plus de 100 000 licenciés répartis dans environ 2 500 clubs sur tout le territoire français. Elle organise les championnats nationaux, qualifie les équipes de France pour les compétitions internationales, et édite les règles officielles du jeu en langue française.

Ce qui frappe lorsqu'on intègre la FFB, c'est le contraste entre la dimension institutionnelle de l'organisation — ses archives, son histoire, son réseau de clubs — et la taille de ses équipes opérationnelles. La direction informatique est composée de quelques personnes, ce qui donne à chaque projet une visibilité et un impact inhabituels pour un stagiaire.

2.1.2 La direction informatique

Mon stage s'est déroulé au sein de la direction informatique, sous la responsabilité de **Laurent Danziger**, chef du département et excellent bridgeur. C'est lui qui a fait le choix de m'accueillir après avoir découvert mon projet universitaire BridgeVision, développé à l'IUT sous la direction de **Hervé Borredon**. Laurent a validé l'orientation technique globale et m'a accordé une liberté de conception rare pour un stagiaire, notamment sur les choix d'architecture et le matériel. Mon maître de stage direct était **Zakaria Oulalite**, chef de projet, qui assurait le suivi quotidien et m'orientait sur les priorités.

Vincent Lamaire, référent technique de la Fédération et bridgeur, faisait également partie des interlocuteurs réguliers. Sa connaissance de l'organisation et sa pratique du jeu ont enrichi les discussions sur les besoins réels du terrain.

Je suis le seul développeur actif sur le projet. L'équipe autour de moi joue un rôle de cadrage et de validation, mais l'ensemble de la conception, du développement et du déploiement repose sur mon travail.

2.2 Le bridge en compétition

2.2.1 Les fondamentaux du jeu

Le bridge est un jeu de cartes à quatre joueurs divisés en deux paires adverses. Par convention, les joueurs sont désignés par les points cardinaux : Nord et Sud forment une paire, Est et Ouest forment l'autre. Le jeu se déroule en deux phases : les enchères (où les joueurs annoncent combien de levées ils espèrent réaliser) et le jeu de la carte (où les 52 cartes sont jouées une à une, en 13 tours appelés *levées*).

En tournoi *duplicate* — le format utilisé dans toutes les compétitions officielles — les mêmes distributions de cartes sont jouées sur plusieurs tables simultanément. Concrètement, dans un tournoi par équipes, chaque équipe s'affronte sur deux tables en parallèle avec des positions opposées : l'Équipe A joue en Nord-Sud à la Table 1 et en Est-Ouest à la Table 2, pendant que l'Équipe B occupe les positions inverses sur les mêmes tables. Le résultat final est la différence entre les scores obtenus sur la même donne dans les deux positions. Ce mécanisme élimine la composante aléatoire — la chance de la distribution — et permet de comparer les performances de façon équitable. Un tournoi mobilise typiquement entre 10 et 20 tables, soit jusqu'à 80 joueurs simultanément. À la fin d'une session, les résultats de toutes les tables sont agrégés pour établir le classement.

2.2.2 Les acteurs clés d'un tournoi

Les opérateurs de tables. Lors d'un tournoi officiel, chaque table est gérée par un **opérateur de table** dont la mission est de saisir manuellement, coup par coup, toutes les cartes jouées dans le logiciel de scoring. Ce travail est continu, répétitif et source d'erreurs : une saisie manquée ou incorrecte fausse le résultat de toute la main. C'est l'impact principal de SmartKibitz : automatiser cette saisie en temps réel, libérant l'opérateur de la contrainte de transcription manuelle et garantissant une fiabilité que l'humain seul ne peut pas maintenir à ce rythme.

Les arbitres. L'arbitre est garant de l'intégrité *et de l'organisation* du jeu. Il est responsable du bon déroulement du tournoi : gestion des rondes, attribution des tables, prise en charge des incidents. Il intervient sur les contestations — une *renonce*, par exemple, c'est jouer un Cœur à la place d'un Carreau, une irrégularité qui peut changer le résultat d'une main entière. Il tranche également les litiges liés à des comportements suspects : les signaux illicites basés sur le temps de réflexion avant de jouer une carte, ou l'orientation intentionnelle donnée à une carte posée sur la table pour communiquer des informations au partenaire. Ces comportements sont difficiles à détecter à l'œil nu, surtout lorsqu'un seul arbitre surveille vingt tables. SmartKibitz lui apporte un outil concret : la possibilité de revoir en vidéo les secondes précédant une contestation, frame

par frame, pour statuer avec certitude.

Les participants. SmartKibitz permet à chaque joueur de revoir sa propre partie a posteriori : l'historique complet des cartes jouées, couplé au replay vidéo, donne accès à une relecture précise de chaque coup joué. C'est une fonctionnalité inexistante aujourd'hui dans les tournois de bridge compétitif.

Les diffuseurs et médias. Le système facilite la retransmission en direct et l'analyse des parties lors des championnats nationaux. L'historique des cartes jouées, associé aux donnes PBN importées, permet d'envisager un suivi analytique automatisé en temps réel, diffusable sur les canaux médias de la FFB.

2.2.3 Une donne de bridge

Pour comprendre le système technique, il faut saisir la notion de **donne** (*board* en anglais). Une donne est une distribution précise des 52 cartes entre les quatre joueurs. Elle est définie par :

— le **donneur** (le joueur qui ouvre les enchères) : Nord, Sud, Est ou Ouest ;
— la **vulnérabilité** : qui est « exposé » à des pénalités plus lourdes (NS, EW, tous, ou personne) ;
— la **distribution** des cartes, exprimée en format PBN (*Portable Bridge Notation*), un standard international.

Exemple de format PBN :

```
[Deal "N:AKJ4.Q5.KJ83.T62 Q862.AK87.5.Q943
        T95.J963.T972.A7 73.T42.AQ64.KJ85"]
```

Listing 2.1. Exemple de donne en format PBN

Chaque session de tournoi comprend entre 24 et 36 donnes. SmartKibitz maintient un registre de ces donnes et les utilise pour valider les cartes détectées par l'IA.

2.3 La mission de stage

2.3.1 La commande initiale

L'objectif de ce stage est la conception d'un **Proof of Concept (POC)** : une version fonctionnelle mais non finalisée, destinée à démontrer la faisabilité technique du projet et à poser les bases d'un déploiement futur à l'échelle des 20 tables.

La demande de départ était formulée de manière assez ouverte : concevoir un système de supervision intelligente capable de reconnaître les cartes posées sur chaque table et de les afficher en temps réel sur un écran central. La direction informatique avait

identifié les Raspberry Pi comme plateforme cible et souhaitait explorer l'utilisation d'une IA de détection d'objets.

Il n'existait aucun système préexistant à reprendre. J'ai démarré de zéro.

Le matériel définitif — les Raspberry Pi, les Camera Module 3, les Hailo AI HAT — n'était pas encore disponible les premiers jours. Pour ne pas perdre de temps, j'ai monté un premier banc d'essai le 5 février avec ce que j'avais sous la main : deux appareils photo Canon posés sur des trépieds au-dessus d'une vraie table de la salle de jeu de la FFB, reliés à mon ordinateur portable. L'objectif n'était pas d'avoir un système fonctionnel, mais de vérifier que la vision par ordinateur pouvait effectivement distinguer les cartes dans les conditions réelles de la salle : l'éclairage, l'angle, la surface du tapis. Ce test a confirmé la faisabilité — et révélé immédiatement les problèmes de reflets et de surexposition qui allaient conditionner les choix d'optique par la suite.

Figure 2.1. Premier banc d'essai — 5 février 2026. Deux appareils photo Canon sur trépieds au-dessus d'une table de bridge de la FFB, avant la réception des Raspberry Pi.

Quatre jours plus tard, le modèle tournait déjà sur ces images. La figure 2.2 montre le premier tableau de bord opérationnel — alors appelé BridgeCam, une version antérieure à SmartKibitz — avec les détections YOLOv8 superposées en temps réel sur les deux flux caméra. Les scores de confiance étaient faibles (beaucoup de faux positifs, cartes détectées plusieurs fois), mais le pipeline fonctionnait bout en bout : caméra, inférence, affichage. C'est à partir de ce résultat brut que j'ai compris quels étaient les vrais problèmes à résoudre — la fusion des deux vues, la déduplication des détections, la robustesse aux cartes qui se chevauchent.

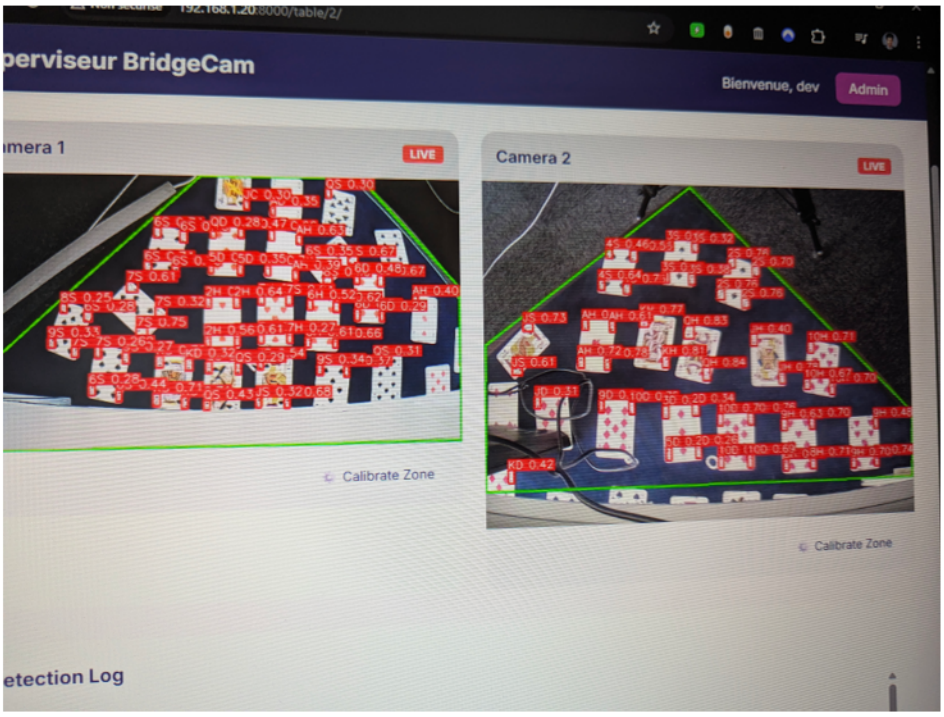


Figure 2.2. Premiers tests modèle en conditions réelles — 9 février 2026. Interface BridgeCam (version prototype), détections YOLOv8 sur deux flux caméra Canon. Les scores de confiance bas (0.28–0.84) reflètent les limitations du modèle face aux cartes superposées et aux reflets du tapis.

2.3.2 Ce que j'ai découvert en arrivant

La réalité du terrain a rapidement enrichi — et parfois compliqué — la commande initiale. Plusieurs contraintes ont émergé dès les premières semaines :

La contrainte air-gap. Les parties de tournoi se déroulent dans des salles sans connexion internet fiable, parfois dans des clubs de province ou des salles municipales louées pour l'occasion. Le système doit fonctionner de façon totalement autonome. Pas de cloud. Pas de services externes. Pas de mise à jour depuis internet pendant le tournoi.

La contrainte matérielle. Vingt Raspberry Pi, vingt paires de caméras, un réseau local dédié. Le matériel doit être installé et opérationnel en moins d'une heure avant chaque tournoi. La procédure doit être reproductible sans compétences techniques particulières.

La contrainte temps réel. Une détection avec 10 secondes de latence est inutile si l'arbitre doit réagir avant que la partie ne progresse. La cible est une latence inférieure à 2 secondes entre le moment où une carte est posée et son affichage sur le tableau de bord.

La contrainte lumière. La luminosité d'une salle de tournoi est variable, parfois médiocre (éclairage fluorescent, ombres portées). Le système IA doit être robuste à ces conditions.

2.3.3 Ma place dans l'organisation

Je travaille seul sur le développement. C'est un avantage — les décisions techniques sont rapides — et une responsabilité importante. Lorsqu'un bug bloque le système, c'est à moi seul de le diagnostiquer et de le corriger.

Les échanges avec Zakaria, Laurent et Vincent alimentent les réflexions d'architecture : choix entre approches, arbitrages fonctionnels, priorisation des fonctionnalités. Ces brainstormings collectifs ont souvent débouché sur de meilleures décisions que ce que j'aurais pris seul. Mais la main sur le clavier, c'est la mienne.

2.4 Le matériel et l'infrastructure

2.4.1 Le réseau physique

Le réseau est conçu comme un réseau local privé totalement isolé. Il repose sur un routeur TP-Link ER605 Omada qui fait office de serveur DHCP et assigne les adresses IP dynamiquement. Tous les appareils sont sur le même sous-réseau : `192.168.38.0/24`.

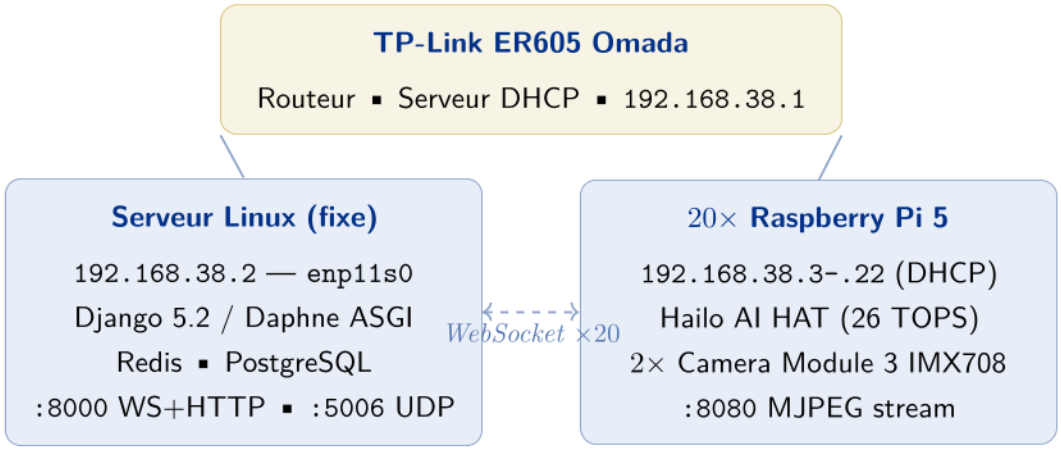


Figure 2.3. Architecture réseau SmartKibitz — réseau air-gapped `192.168.38.0/24`

2.4.2 Les nœuds edge

Le choix du Raspberry Pi 5 comme plateforme de déploiement n'était pas acquis d'avance. Au démarrage du projet, plusieurs architectures étaient envisageables. La discussion avec Laurent Danziger et Zakaria Oulalite a progressivement convergé vers le Raspberry Pi pour des raisons concrètes : montage modulaire rapide sur chaque table, alimentation par câble réseau unique (PoE), maintenance sur place sans outillage spécifique, communauté et documentation abondantes, et coût unitaire maîtrisé. Un

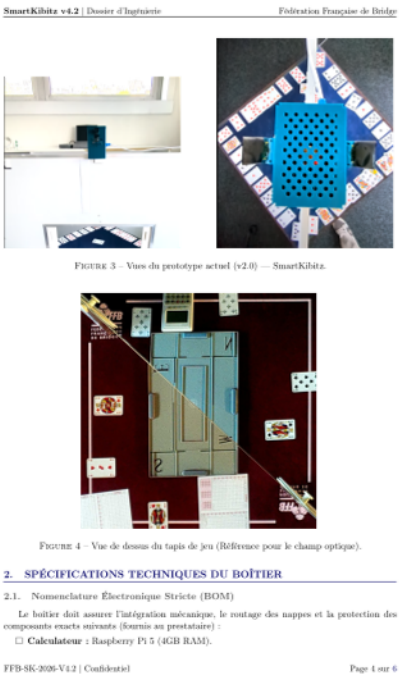
Raspberry Pi peut être remplacé en moins de cinq minutes sans formation particulière — ce qui compte dans un contexte de tournoi.

Chaque nœud edge est composé de :

- un **Raspberry Pi 5** (4 cœurs Cortex-A76 à 2,4 GHz, 8 Go RAM), sous Raspberry Pi OS Lite 64-bit Bookworm ;
- un **Hailo AI HAT** — accélérateur neuronal offrant 26 TOPS (trillions d'opérations par seconde) pour une consommation inférieure à 8 W ;
- deux **Camera Module 3** (Sony IMX708, 12 Mpx, grand angle) montées en vis-à-vis au-dessus de la table ;
- un **boîtier imprimé en 3D** (PETG/ASA prototype, PA12 SLS en série), conçu sur mesure pour s'installer sur le séparateur central de la table de bridge sans modification du mobilier.

Le Pi est alimenté en *Power over Ethernet* (PoE) : un seul câble réseau suffit pour transporter alimentation électrique et connexion réseau, réduisant le câblage à une seule prise par table.

(a) Profil de montage sur le séparateur de table. Cahier des charges rédigé en vue des futures impressions série.



(b) Prototype boîtier et vue de dessus du tapis de jeu (champ optique des caméras).

Figure 2.4. SmartKibitz v4.2 — Boîtier de montage sur table de bridge.

2.4.3 Le serveur central

Le serveur est un PC Linux (Ubuntu/Debian) à IP fixe (`192.168.38.2`). Il héberge le serveur web Django, la base de données PostgreSQL, le broker de messages Redis, et

les services annexes (scanner réseau, provisionneur automatique, serveur de diffusion UDP). C'est lui qui reçoit toutes les détections des nœuds edge, les stocke, et alimente le tableau de bord.

2.5 Avancement au terme du stage

Au 15 mai 2026, deux tables sur vingt sont pleinement opérationnelles. Le pipeline complet — de la capture caméra à l'affichage sur le tableau de bord — fonctionne de bout en bout. Le provisionnement automatique a été validé sur plusieurs Raspberry Pi. La mise à jour OTA a été utilisée en conditions réelles à plusieurs reprises.

Les dix-huit tables restantes n'ont pas encore été équipées, faute d'un volume de matériel disponible. L'infrastructure logicielle est prête à les accueillir sans modification.

Chapitre 3

Analyse du besoin et conception

3.1 Recueil des besoins

3.1.1 Besoins fonctionnels

Avant d'écrire la première ligne de code, j'ai listé précisément ce que le système devait faire :

1. **Détecter les cartes** posées sur chaque table en temps réel, avec une latence maximale de 2 secondes entre le dépôt de la carte et son affichage sur le tableau de bord.
2. **Afficher l'état de toutes les tables** simultanément sur une interface web accessible depuis n'importe quel navigateur du réseau local.
3. **Maintenir un historique** des cartes jouées par table et par donne, y compris en cas de coupure réseau temporaire.
4. **Alerter** l'arbitre en cas d'anomalie détectée (carte non conforme à la donne, erreur système, perte de connexion d'un nœud).
5. **Permettre la collecte de données** pour réentraîner le modèle IA sur des images réelles.
6. **Déployer et mettre à jour** les nœuds edge sans intervention manuelle sur le matériel.

3.1.2 Besoins non fonctionnels

Au-delà des fonctionnalités, plusieurs contraintes d'ordre technique et opérationnel s'imposaient :

— **Air-gap** : aucune dépendance à Internet. Toutes les bibliothèques, tous les fichiers, tout le code doivent être disponibles localement.
— **Résilience** : le système doit survivre à la perte de connexion d'un ou plusieurs Raspberry Pi sans planter le serveur central.
— **Déployabilité** : l'installation d'un nouveau Pi doit pouvoir se faire en moins de 10 minutes, de préférence sans intervention manuelle sur le Pi lui-même.
— **Maintenabilité** : le code déployé sur les Pi doit être modifiable et redéployable sans reflasher les cartes SD.

— **Scalabilité** : l'architecture doit supporter 20 connexions WebSocket simultanées et persistantes.

3.2 Choix d'architecture : les décisions clés

3.2.1 Framework serveur : Django plutôt que FastAPI ou Node.js

Le premier choix fondamental a été celui du framework backend. Trois candidats se dessinaient :

Framework	ORM intégré	WebSockets	Admin UI	Async natif
Django 5.2	✓	via Channels	✓	ASGI
FastAPI	×	via Starlette	×	✓
Node.js (Express)	×	natif	×	✓

J'ai retenu Django pour plusieurs raisons. L'ORM intégré évite de coupler un framework à une couche d'accès aux données séparée. Django Channels est une extension mature qui ajoute le support WebSocket avec peu de friction. Django Admin offre gratuitement une interface de gestion — une économie de développement significative.

Un facteur non négligeable s'est ajouté : Django est le framework que j'avais appris à l'IUT d'Orsay avec Hervé Borredon, et que j'avais utilisé dans BridgeVision. La maîtrise préalable d'un outil réduit la dette d'apprentissage au profit de la vitesse d'exécution — ce qui comptait dans un stage de 3,5 mois.

Le contre-argument contre Django est sa lourdeur pour une application pure WebSocket. C'est vrai. Mais pour ce cas d'usage qui combine API HTTP, WebSockets persistants, base de données relationnelle et interface d'administration, le ratio est favorable.

3.2.2 ASGI vs WSGI : pourquoi le mode synchrone ne suffit pas

WSGI est comme un standardiste qui raccroche après chaque appel et attend la sonnerie suivante. ASGI est un standardiste qui peut garder vingt lignes ouvertes simultanément sans jamais raccrocher.

Le protocole WSGI (Web Server Gateway Interface), utilisé par Django dans sa configuration classique, est synchrone : il traite une requête à la fois par thread. Pour des pages web classiques, c'est suffisant. Pour vingt connexions WebSocket permanentes — une par table — c'est une impasse : chaque connexion maintient un canal ouvert indéfiniment, ce qui bloquerait les threads WSGI.

ASGI (Asynchronous Server Gateway Interface), supporté par Django depuis la version 3.1 et servi par Daphne, résout ce problème en permettant des connexions longue durée gérées de façon asynchrone. Le serveur peut maintenir 20 WebSockets ouverts simultanément avec un seul processus.

3.2.3 IA embarquée vs GPU centralisé

Un GPU central, c'est envoyer toutes les photos de surveillance de la ville dans un bureau central pour les analyser. L'IA embarquée, c'est installer un analyste dans chaque immeuble. Moins de trafic réseau, plus de résilience.

Le choix entre inférence sur chaque Pi et inférence centralisée sur un GPU serveur n'a pas été immédiat. Voici la comparaison que j'ai effectuée :

Critère	Hailo AI HAT (edge)	GPU serveur (centralisé)
Latence	∼50 ms (local)	∼200 ms (réseau + file)
Consommation	5 W par Pi	100–300 W
Résilience	Si serveur tombe, Pi continue	Single Point of Failure
Coût	70 EUR × 20 = 1 400 EUR	500–2 000 EUR × 1
Bande passante	Faible (JSON uniquement)	Élevée (frames vidéo en entrée)

La résilience a été l'argument décisif : si le serveur central rencontre un problème pendant un tournoi, les Pi doivent continuer à détecter les cartes localement. L'option GPU centralisé crée un point de défaillance unique inacceptable dans un contexte de compétition officielle.

Le système prévoit néanmoins un mode de fallback GPU serveur (variable d'environnement `USE_SERVER_GPU=1`) pour les tests et les cas où le Hailo HAT n'est pas disponible.

3.2.4 Découverte réseau : UDP Broadcast plutôt que configuration statique

Le serveur crie dans le couloir : « Je suis là, port 8000 ! » Les Pi qui écoutent notent l'adresse et viennent frapper à la porte.

Les Raspberry Pi obtiennent leurs adresses IP de façon dynamique (DHCP). Configurer manuellement l'adresse du serveur sur vingt cartes SD est une source d'erreurs et d'incompatibilités lors des mises à jour. La solution retenue est un mécanisme de diffusion UDP : le serveur émet toutes les deux secondes un paquet broadcast signé (HMAC-SHA256) sur le port 5006. Les Pi à l'écoute extraient l'adresse IP du serveur de ce message et s'y connectent.

Cette approche est résistante aux changements d'IP du serveur (si le serveur redémarre avec une IP différente, les Pi se reconnectent automatiquement) et ne nécessite aucune configuration sur les Pi au-delà de l'activation du service.

3.2.5 Event Sourcing pour les cartes jouées

Un journal de bord de marin : on ne rature jamais ce qui est écrit. Si on veut l'état actuel du voyage, on relit le journal depuis le début.

La question de la représentation des cartes jouées en base de données semblait simple au premier abord : une colonne *cartes_jouées* contenant l'état courant. Mais ce schéma naïf posait un problème : si la connexion entre le Pi et le serveur est coupée en milieu de main, les cartes détectées localement pendant la coupure sont perdues, ou écrasées au moment de la reconnexion.

La solution est l'event sourcing : le champ `played_cards_log` est une liste append-only de chaînes de la forme `"SEAT:CARD"` (par exemple `"N:AS"` pour l'As de Pique joué par Nord). Jamais d'écrasement, jamais de suppression. Lors d'une reconnexion, le Pi envoie son historique local, et le serveur fusionne les deux listes par déduplication :

```
def merge_edge_history(server_log, edge_log):
    server_set = set(server_log)
    new_from_edge = [c for c in edge_log if c not in server_set]
    return server_log + new_from_edge
```

Listing 3.1. Fusion d'historiques par déduplication

Ce mécanisme est idempotent : si le même événement arrive plusieurs fois (reconnexion multiple), le résultat est identique. Aucune carte n'est perdue, aucun doublon n'est créé.

3.2.6 Zero-Touch Provisioning vs Golden Image

La méthode classique pour déployer du code sur un Raspberry Pi est de créer une « Golden Image » — une image SD préconfigurée que l'on clone sur chaque carte. C'est simple pour quelques Pi, mais problématique à l'échelle :

— flasher 20 cartes SD = 20 × 10 minutes de travail répétitif ;
— si le code change, il faut reflasher les 20 cartes ;
— la gestion des versions sur 20 cartes physiques est cauchemardesque.

L'approche retenue est radicalement différente : chaque Pi démarre avec Raspberry Pi OS Lite en version stock (firmware officiel, SSH activé, aucune configuration). Le serveur détecte automatiquement les nouveaux Pi sur le réseau via un scanner ARP, se connecte en SSH, déploie le code et active le service. Cinq minutes par Pi, aucune intervention manuelle sur le matériel.

3.3 Modèle de données

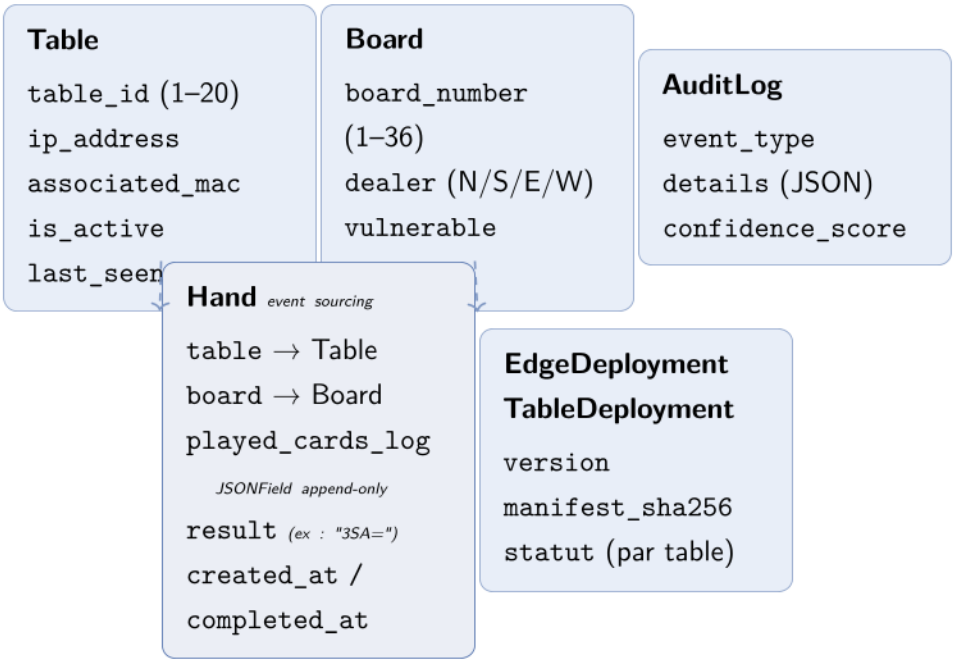


Figure 3.1. Modèle de données SmartKibitz (`core/models.py`)

3.4 Le protocole WebSocket

La communication temps réel repose sur trois types de connexions WebSocket :

— `ws://serveur:8000/ws/table/{id}/` — canal Pi↔Serveur (`TableConsumer`)
— `ws://serveur:8000/ws/dashboard/` — canal Serveur↔Navigateur (`DashboardConsumer`)

Table 3.1. Messages WebSocket principaux

Direction	Type	Contenu
Pi –> Serveur	`card_event`	cartes détectées (seat, card, confidence)
Pi –> Serveur	`stats`	CPU%, RAM%, temp°C, FPS
Pi –> Serveur	`alert`	message d'alerte
Pi –> Serveur	`sync`	historique local (reconnexion)
Pi –> Serveur	`setup_log`	messages de démarrage
Serveur –> Dashboard	`initial_state`	état complet à la connexion
Serveur –> Dashboard	`dashboard_update`	nouvelles cartes (diff d'état)
Serveur –> Dashboard	`table_status_update`	Pi connecté/déconnecté
Serveur –> Dashboard	`table_stats_update`	télémétrie (CPU/RAM/temp/FPS)
Dashboard –> Serveur	`command`	reboot, record, calibration...

3.5 Architecture générale

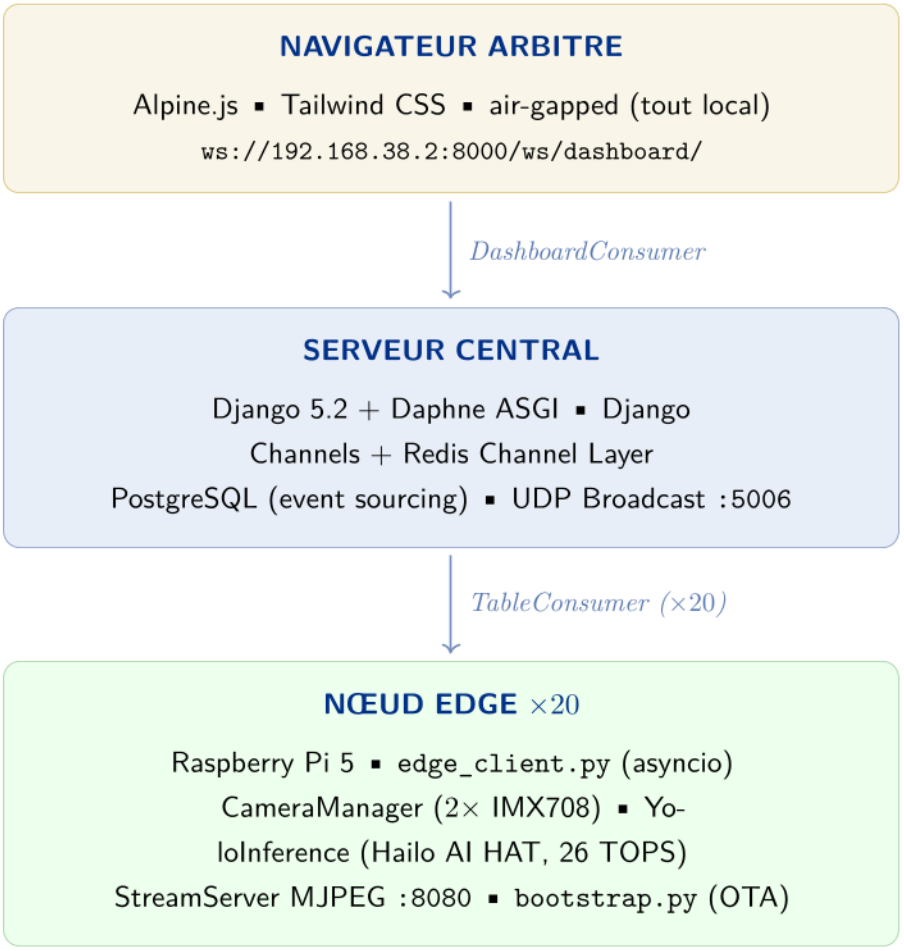


Figure 3.2. Architecture globale SmartKibitz

Chapitre 4

Développement et implémentation

4.1 Le serveur central Django ASGI

4.1.1 Structure du projet Django

Le serveur est une application Django standard organisée autour d'un module applicatif `core/` qui contient l'ensemble de la logique métier. Le fichier `asgi.py` configure Daphne pour router les requêtes HTTP classiques vers le gestionnaire Django standard, et les connexions WebSocket vers le routeur Django Channels.

```
application = ProtocolTypeRouter({
    "http": get_asgi_application(),
    "websocket": AuthMiddlewareStack(
        URLRouter(core.routing.websocket_urlpatterns)
    ),
})
```

Listing 4.1. Routage ASGI dans bridge_server/asgi.py

Le routeur WebSocket associe deux URLs à deux consumers :

- `ws/table/{table_id}/` –> `TableConsumer`
- `ws/dashboard/` –> `DashboardConsumer`

4.1.2 Les consumers WebSocket

Un *consumer* Django Channels est l'équivalent d'une vue Django, mais pour les WebSockets. Il gère les événements de connexion, de déconnexion et de réception de messages.

TableConsumer — le canal Pi vers serveur

Le `TableConsumer` gère la connexion de chaque Raspberry Pi. À la connexion, il met à jour la table en base de données (`is_active = True`), rejoint un groupe de canaux nommé `table_{id}`, et envoie un accusé de réception au Pi.

La partie la plus délicate est le traitement du message `card_event` :

```
async def handle_card_event(self, data):
```

```
    hand = await get_or_create_hand(self.table_id, data["board_id"
        ])
    new_cards = await process_card_events(hand, data["cards"])
    if new_cards:
        await self.channel_layer.group_send(
            "dashboard", {
                "type": "dashboard_update",
                "table_id": self.table_id,
                "new_cards": new_cards,
            }
        )
```

Listing 4.2. Traitement d'un événement carte (simplifié)

Le **state diffing** est la clé de l'efficacité : au lieu de renvoyer l'état complet de la main à chaque frame, on n'envoie que les cartes nouvellement détectées par rapport à l'état précédent. Cela réduit considérablement le trafic WebSocket vers le tableau de bord.

Le consumer implémente également un **watchdog** : une tâche asyncio qui vérifie toutes les 60 secondes qu'un message de type `stats` a bien été reçu. Si ce n'est pas le cas, la connexion est fermée proprement et le tableau de bord est notifié. Ce mécanisme prévient les *connexions fantômes* — des Pis qui semblent connectés sur le dashboard mais ne répondent plus physiquement (câble débranché, Pi planté).

DashboardConsumer — le canal serveur vers navigateur

Le `DashboardConsumer` est plus simple : il rejoint le groupe `dashboard` et relaye les messages reçus depuis les `TableConsumer` vers tous les navigateurs connectés. À la connexion initiale, il envoie l'état complet de toutes les tables (message `initial_state`) pour synchroniser la vue du navigateur.

4.1.3 Services partagés

Pour éviter la duplication de logique entre les consumers et les vues HTTP, deux fonctions utilitaires sont centralisées dans `core/services.py` :

```
async def get_or_create_hand(table_id, board_id):
    """Retourne la main active ou en crée une nouvelle."""
    ...

def is_session_fresh():
    """Vérifie si le fichier session_start.txt est présent."""
    return Path("core/session_start.txt").exists()
```

Listing 4.3. Helpers partagés dans core/services.py

4.1.4 Le tableau de bord Alpine.js

React, c'est une usine à gaz avec des convoyeurs et des robots. Alpine.js, c'est un couteau suisse : tu le glisses dans ta poche, il fait le travail, et il n'a pas besoin d'une prise de courant.

L'interface graphique est construite avec Alpine.js (15 Ko, un seul fichier JavaScript) et Tailwind CSS (mode runtime, un seul fichier). Le choix est dicté par la contrainte air-gap : pas de connexion internet, pas de npm, pas de bundler. Un seul fichier `dashboard_app.js` contient toute la logique réactive.

La grille principale affiche 20 cases, une par table, organisées en 5 colonnes × 4 rangées. Chaque case représente une table de bridge sous forme de **plan architectural** : un carré SVG avec quatre rectangles figurant les chaises aux positions cardinales (N, S, E, O) et les labels de direction. À l'intérieur, le tapis de jeu (surface rouge) est toujours visible — à pleine intensité quand le Pi est connecté, fortement atténué (opacité 20%, désaturé) quand il ne l'est pas. Sur le tapis actif s'affichent :

— le numéro de table en haut ;
— le score $x/52$ en bas, en amber tant que la main est en cours, en vert à 52/52 ;
— un overlay SVG superposant les cartes détectées en temps réel avec leur position sur le tapis ;
— des indicateurs d'état : badge REC (mode collecte), coche verte (main terminée), point rouge clignotant (alerte).

Sous l'icône de table, une ligne d'information affiche la dernière carte détectée en notation française (V♡, D♢, R♠…) et les métriques système (FPS et température).

Le clic sur une case ouvre une modale en deux colonnes : à gauche le flux vidéo MJPEG en direct depuis le port 8080 du Pi et la chronologie des cartes jouées, à droite les statistiques système (CPU, RAM, température, FPS) et les journaux de démarrage.

4.2 Le nœud edge Raspberry Pi 5

4.2.1 L'orchestrateur edge_client.py

Le nœud edge est centré sur `edge_client.py`, un orchestrateur asyncio qui démarre et coordonne tous les sous-systèmes du Pi :

```
async def main():
    camera_mgr = CameraManager()
    stream_srv = StreamServer(camera_mgr)

    await asyncio.gather(
        camera_mgr.run(),
```

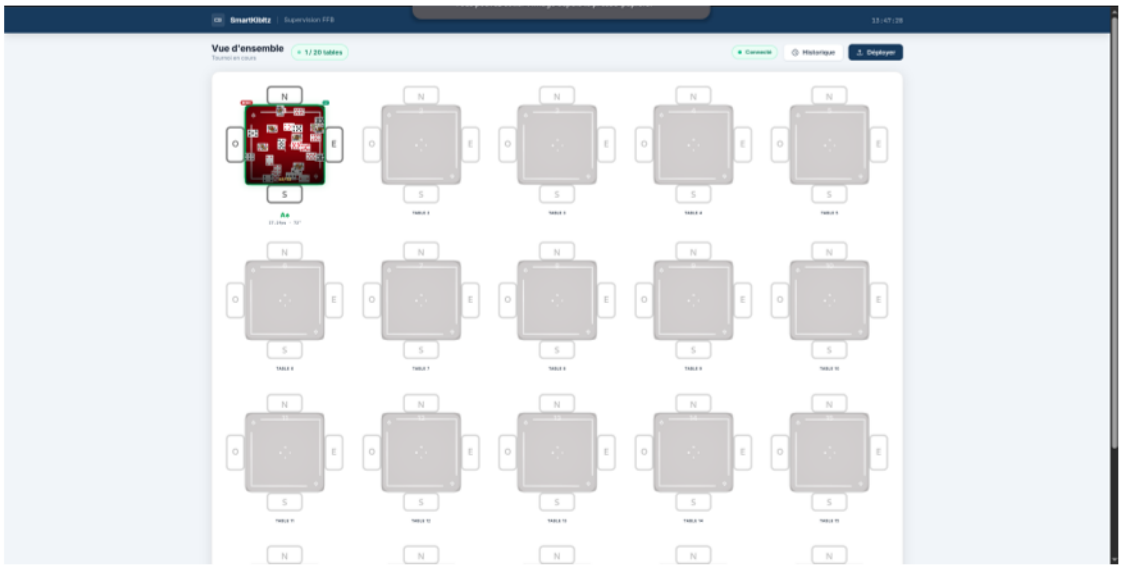


Figure 4.1. Vue d'ensemble du Dashboard (Mode Sombre) — L'interface est optimisée pour une consultation prolongée lors des tournois nocturnes, tout en conservant la visibilité critique sur l'état des tables et du réseau.

```
        stream_srv.run(),
        connect_with_backoff(camera_mgr),
    )
```

Listing 4.4. Démarrage des sous-systèmes dans edge_client.py (simplifié)

La connexion au serveur est gérée avec un **backoff exponentiel** : si le serveur n'est pas disponible, le Pi attend 3 secondes avant la première tentative, 6 secondes après un deuxième échec, 12 secondes après un troisième, et ainsi de suite jusqu'à un plafond de 60 secondes. Dès que la connexion est rétablie, le compteur se remet à zéro.

Problème évité : le Thundering Herd

Si le serveur redémarre, les 20 Pi tentent de se reconnecter simultanément. Sans backoff, cette avalanche de connexions simultanées surcharge le serveur à l'exacte minute où il vient de démarrer. Le backoff exponentiel étale les reconnexions dans le temps, éliminant ce risque.

4.2.2 CameraManager : la gestion duale des caméras

Chaque Pi gère deux caméras. Le `CameraManager` lit les flux vidéo et maintient deux buffers distincts :

— `_latest_jpeg` : le dernier frame JPEG brut, utilisé par le serveur MJPEG (visualisation) ;
— `_latest_frames` : les derniers frames numpy (tableaux de pixels), utilisés par le pipeline IA.

Cette séparation est importante : le serveur MJPEG a besoin d'images fréquentes et légères (fluide pour l'œil humain), tandis que le pipeline IA a besoin d'images traitées et peut se permettre une fréquence plus faible (traiter chaque frame à 30 fps serait inutile et coûteux).

Chaîne de fallback pour l'accès caméra

Les Raspberry Pi peuvent exécuter différentes versions du système d'exploitation, et les outils d'accès caméra varient selon la version :

1. **rpicam-vid** (libcamera moderne, Pi OS Bookworm) — tentative en premier
2. **raspivid** (stack legacy, Pi OS Buster/Bullseye) — si rpicam-vid échoue
3. **OpenCV V4L2** (`/dev/videoN`) — fallback universel, toujours disponible

Chaque caméra est indépendante dans sa chaîne de fallback : si une caméra utilise rpicam-vid et que l'autre utilise V4L2, le système fonctionne parfaitement. Un watchdog interne relance automatiquement la capture si aucun frame n'est reçu pendant 15 secondes.

4.2.3 Le pipeline IA

Acquisition et fusion des images

Les deux Camera Module 3 sont montées au-dessus de la table, l'une visant la moitié Nord/Est de la table, l'autre la moitié Sud/Ouest. La caméra 1 est physiquement montée à l'envers (rotation 180°).

La fusion des deux images en une seule image composite de 960 × 960 pixels suit le processus suivant :

Le rail de scoring physique (barre brillante centrale entre les deux moitiés de la table) apparaît comme une bande lumineuse au centre de l'image composite. Ce n'est pas un artefact corrigeable : c'est un objet physique réel. Une option de masquage est disponible mais désactivée par défaut.

Calibration triangulaire (mode avancé)

Pour les tables où la géométrie de montage des caméras est précisément connue, un mode de calibration plus précis est disponible. L'arbitre clique sur 3 coins par caméra dans l'interface web (onglet « Fusion »). Le Pi calcule alors des transformations affines individuelles par caméra et fusionne les images par triangles au lieu d'un empilement vertical.

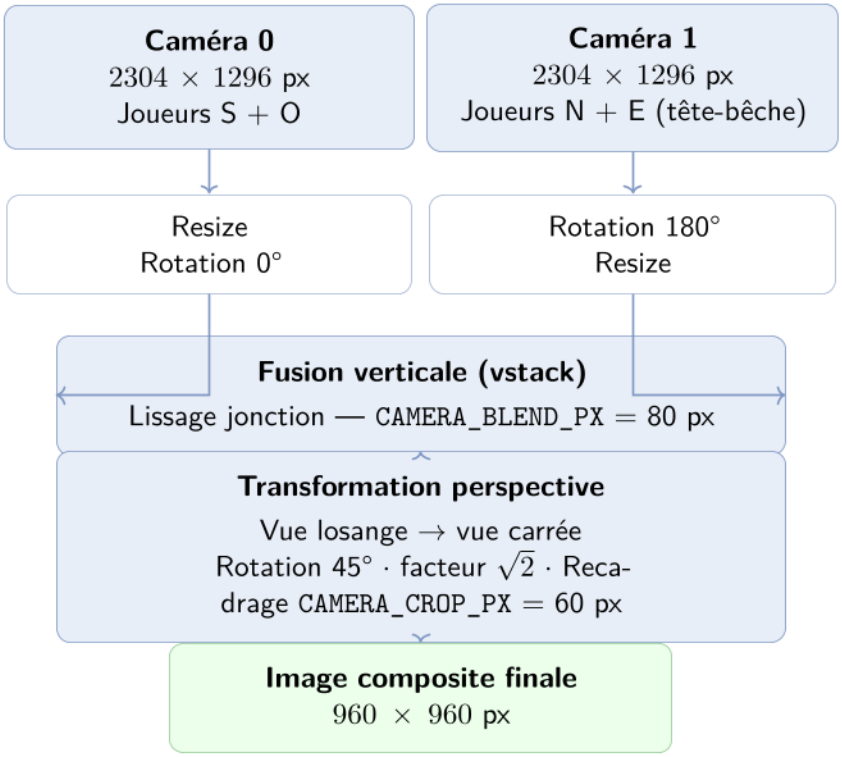


Figure 4.2. Pipeline de fusion des images caméra

Inférence YOLOv8m sur Hailo AI HAT

Le Hailo AI HAT est un cerveau spécialisé : il ne sait faire qu'une seule chose, reconnaître des objets dans des images, mais il le fait vite et pour une consommation dérisoire.

Le modèle utilisé est un **YOLOv8m** (medium) compilé au format `.hef` (Hailo Executable Format). Ce format est le résultat de la compilation du modèle ONNX par le SDK Hailo, qui optimise le graphe de calcul pour l'architecture matérielle spécifique de la puce Hailo-8.

Le modèle détecte 52 classes correspondant aux 52 cartes d'un jeu standard :

— 13 valeurs : As, 2, 3, ..., 10, Valet, Dame, Roi
— 4 couleurs : Pique (S), Cœur (H), Carreau (D), Trèfle (C)

L'origine du modèle mérite d'être précisée : il a été initialement entraîné dans le cadre d'un projet scolaire antérieur au stage, nommé **BridgeVision**. Ce premier entraînement s'est appuyé sur des jeux de données publics de cartes à jouer. Pendant le stage, j'ai utilisé le Mode Record (voir section 4.2.4) pour collecter des images réelles de parties à la FFB, dans les conditions d'éclairage et d'angle réelles d'un tournoi. Ces nouvelles données ont permis de **raffiner le modèle** pour qu'il soit plus robuste aux conditions spécifiques de la salle de tournoi.

L'inférence produit des boîtes englobantes (*bounding boxes*) avec un score de confiance.

Seules les détections avec un score supérieur à 0,15 sont retenues. Un mécanisme de **stabilisation sur deux frames consécutives** filtre les faux positifs : une carte n'est considérée comme détectée que si elle apparaît dans deux frames successifs avec la même identité.

4.2.4 Mode Record : collecte du dataset

La performance du modèle IA dépend directement de la qualité des données d'entraînement. Pour améliorer le modèle au fil du temps, SmartKibitz intègre un mode de collecte de données directement depuis les Pi.

Lorsque l'arbitre active le Mode Record depuis le tableau de bord, le Pi bascule en mode collecte : l'inférence IA est mise en pause, et le Pi enregistre des frames JPEG à 0,5 fps et les envoie au serveur. Ces images sont sauvegardées dans `media/dataset_raw/table_{id}/cam_{n}/`.

Après le tournoi, ces images peuvent être importées dans Roboflow pour annotation, puis utilisées pour réentraîner YOLOv8m. Une fois le nouveau modèle entraîné, il est compilé au format `.hef` par le SDK Hailo et redéployé sur les Pi via OTA Hot-Swap.

4.2.5 Replay : rejouer un coup ou une partie complète

Le Mode Record a naturellement ouvert la porte à une fonctionnalité que je n'avais pas anticipée dans la conception initiale : le replay. Une fois une session d'enregistrement terminée et les frames stockées sur le serveur, il devient possible de les relire frame par frame depuis le navigateur.

SmartKibitz distingue deux modes de replay accessibles depuis le tableau de bord, dans l'onglet « Historique ».

Replay de coup (historique des mains)

Le premier mode ne nécessite aucun enregistrement vidéo. Il exploite directement le champ `played_cards_log` du modèle `Hand` — le journal append-only de toutes les cartes jouées. L'endpoint `GET /api/history/` retourne la liste des mains enregistrées (filtrables par table). L'endpoint `GET /api/history/{id}/` retourne le détail d'une main : les cartes jouées regroupées par siège (Nord, Sud, Est, Ouest) et la séquence chronologique complète.

C'est utile pour un arbitre qui veut vérifier a posteriori quelles cartes ont été jouées à la table 3 lors de la donne 17, sans avoir besoin d'une vidéo.

Replay vidéo frame par frame

Le second mode est plus ambitieux. Lorsqu'une session a été enregistrée (Mode Record actif), le serveur a stocké des frames JPEG dans `media/dataset_raw/table_{id}/`.

Le tableau de bord exploite ces frames pour offrir un player vidéo :

— un slider permet de naviguer dans le temps (de la frame 0 à la dernière) ;
— le bouton Play/Pause avance automatiquement à vitesse réglable (0,5× à 8×) ;
— une **timeline** superposée indique, par des repères colorés, les frames où une carte a été détectée — une couleur par siège (N, E, S, O) ;
— au frame courant, les cartes détectées avec leur score de confiance YOLO sont affichées en regard de l'image.

Ces données temporelles viennent du modèle `CardDetectionEvent`, qui associe chaque détection à un `frame_index` dans la session. L'endpoint `GET /api/sessions/{id}/timeline/` retourne l'ensemble de ces événements pour une session, permettant au front-end de construire la timeline sans re-parcourir les frames.

```
path('api/sessions/', views.sessions_list),
path('api/sessions/<int:session_id>/frame/<int:frame_index>/',
     views.session_frame),
path('api/sessions/<int:session_id>/timeline/',
     views.session_timeline),
```

Listing 4.5. Endpoints replay dans core/urls.py

L'endpoint `session_frame` sert directement le fichier JPEG depuis le disque via `FileResponse`, sans passer par la base de données. C'est important pour la fluidité du replay : naviguer dans une session de plusieurs milliers de frames doit rester réactif.

Ce mode de replay est l'équivalent d'une *Video Assistance Referee* (VAR) pour le bridge : un arbitre peut revoir les secondes qui précèdent une contestation et vérifier si une carte a bien été posée dans le bon ordre, avec le score de confiance de l'IA à l'appui.

4.3 Zero-Touch Provisioning

Le provisionnement automatique est une des fonctionnalités dont je suis le plus fier, précisément parce qu'elle résout un problème concret qui aurait rendu le déploiement à 20 tables extrêmement pénible.

La durée totale pour provisionner un nouveau Pi est d'environ 5 minutes, sans aucune intervention manuelle sur le Pi lui-même. Un technicien qui branche simplement le câble réseau peut provisionner les 20 tables pendant qu'il installe le matériel sur les tables.

La table est assignée par MAC address : une fois qu'un Pi a été associé à la table 5, il sera toujours la table 5, même s'il redémarre ou change d'IP. Cette association permanente est stockée dans le champ `associated_mac` du modèle `Table`.

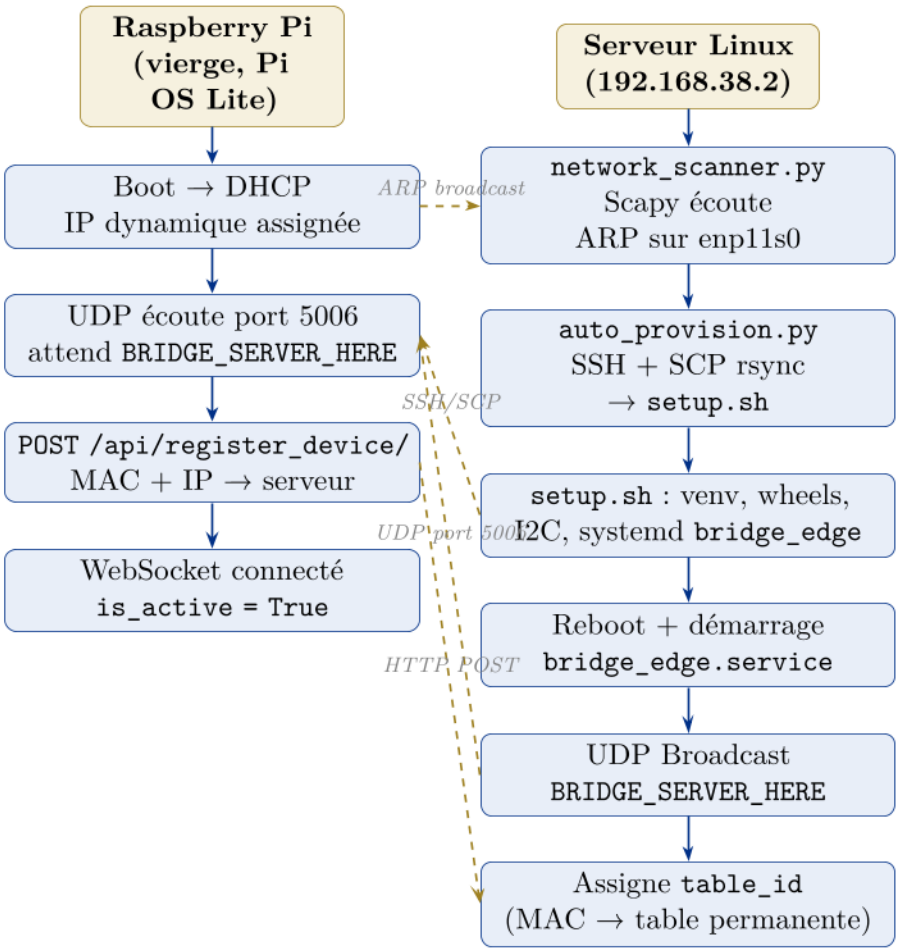


Figure 4.3. Diagramme de flux — Zero-Touch Provisioning (ZTP)

4.4 OTA Hot-Swap

Mettre à jour le code d'un Pi sans OTA, c'est comme changer une roue en arrêtant la voiture, démontant le moteur, puis remontant tout. L'OTA Hot-Swap change la roue sans même ralentir.

Le système OTA (*Over-The-Air*) permet de mettre à jour le code de tous les Pi depuis le tableau de bord, sans y toucher physiquement et sans interrompre leur fonctionnement (ou en les redémarrant de façon contrôlée).

4.4.1 Le mécanisme de symlink atomique

```
# Téléchargement dans un répertoire temporaire
download_to(manifest, tmp_dir)

# Vérification SHA256 de chaque fichier
verify_checksums(manifest, tmp_dir)

# Échange atomique : l'ancien répertoire courant
```

```
# reste intact jusqu'au dernier instant
os.rename(tmp_dir, current_dir)

# edge_client.py utilisera le nouveau code
# au prochain démarrage
```

Listing 4.6. Échange atomique dans bootstrap.py

L'opération `os.rename()` est atomique au niveau du système de fichiers : soit elle réussit complètement, soit elle échoue sans laisser d'état intermédiaire. Il n'y a jamais de moment où le Pi a un code partiellement mis à jour.

4.4.2 Processus de déploiement

1. Le développeur pousse des modifications dans `edge_node/`.
2. Il déclenche la création d'un nouveau `EdgeDeployment` depuis le tableau de bord.
3. Le serveur génère un manifeste JSON (liste des fichiers + checksums SHA256) et archive le code.
4. Le serveur notifie les Pi via WebSocket : « une nouvelle version est disponible ».
5. Chaque Pi télécharge l'archive, vérifie les checksums, effectue le symlink atomique.
6. Le Pi confirme le succès (ou l'échec) au serveur via WebSocket.

4.5 Sécurité

La nature air-gapped du réseau réduit la surface d'attaque, mais n'élimine pas tous les risques. Plusieurs mesures de sécurité ont été intégrées au fil du développement (certaines suite à un audit de code formalisé) :

— **SECRET_KEY cryptographique** : la clé secrète Django est générée par `secrets.token_hex(32)` et stockée dans `.env`, jamais en dur dans le code. À la v1.3.3, la clé était littéralement la chaîne `"ramadan"` — anecdote embarrassante mais instructive.

— **subprocess vs os.system** : toutes les commandes shell utilisent `subprocess.run()` avec une liste d'arguments, jamais `os.system()` avec une chaîne interpolée, ce qui élimine les risques d'injection shell.

— **Validation du format MAC** : le endpoint `/api/register_device/` valide le format de l'adresse MAC avec une expression régulière avant de l'insérer en base de données.

— **HMAC-SHA256 sur UDP Broadcast** : les messages de découverte réseau sont signés pour empêcher un attaquant sur le réseau local de rediriger les Pi vers un serveur malveillant.

— **Limite de taille des requêtes** : le endpoint d'inférence GPU est limité à 10 Mo pour éviter les attaques par saturation mémoire.

Chapitre 5

Tests, débogage et qualité

5.1 Tests unitaires

La suite de tests unitaires couvre les composants les plus critiques du serveur : la fusion d'historiques de cartes, la validation du format PBN, les helpers de services, et les endpoints HTTP principaux. Ces tests s'exécutent avec `python manage.py test` et utilisent une base de données SQLite en mémoire pour l'isolation.

La couverture est intentionnellement concentrée sur la logique métier (event sourcing, déduplication) plutôt que sur l'infrastructure (consumers WebSocket, déploiement OTA), dont les tests relèvent davantage de tests d'intégration en conditions réelles.

5.2 Tests d'intégration terrain

Les tests terrain sont ceux qui ont le plus de valeur dans ce projet, et aussi les plus difficiles à automatiser. Ils consistent à déployer le système sur les deux Raspberry Pi disponibles, à simuler des parties de bridge, et à vérifier le comportement du tableau de bord en temps réel.

La méthodologie que j'ai adoptée est simple : avant chaque modification significative du code edge, je déploie via OTA, observe le comportement pendant 5 à 10 minutes de simulation, et note les anomalies. Cette boucle courte m'a permis de détecter rapidement les régressions.

5.3 Les six bugs les plus instructifs

5.3.1 Bug #1 — Le crash réseau de la Fédération

Symptôme

Quelques minutes après avoir branché deux caméras IP sur le réseau interne de la FFB et lancé un test de streaming vidéo, toute la Fédération perd sa connexion internet. Les collègues sont déconnectés de leurs outils de travail. L'incident dure plusieurs minutes.

Diagnostic. Je n'avais pas encore de réseau dédié. J'ai naïvement branché mes caméras sur le réseau interne de la FFB, qui partage la même connexion internet que les bureaux. Un flux vidéo brut 4K (environ 50 Mbit/s par caméra) a saturé la bande passante de la connexion internet de toute la Fédération.

Fix. Déployer un switch réseau dédié, physiquement isolé du réseau principal de la FFB. Ce switch ne contient que le serveur SmartKibitz et les Raspberry Pi — aucune connexion vers l'extérieur.

Leçon. Cet incident est à l'origine directe de l'architecture air-gap du projet. Ce n'était pas une décision de conception abstraite : c'était une nécessité concrète découverte de la façon la plus inconfortable qui soit. L'isolement réseau n'est pas qu'une bonne pratique de sécurité — c'est une nécessité opérationnelle quand on déploie des flux vidéo hauts débits dans un environnement partagé.

5.3.2 Bug #2 — Le serveur DHCP parasite

Symptôme

Lors d'un test de configuration réseau, certains ordinateurs de la FFB perdent leur accès à Internet. Les collègues ne comprennent pas pourquoi.

Diagnostic. J'avais activé un serveur DHCP sur mon serveur Linux pour tester la distribution automatique d'adresses IP. Ce serveur répondait à toutes les requêtes DHCP sur l'interface réseau, y compris celles des appareils du réseau de la FFB qui demandaient une adresse IP au routeur de la fédération. Mon serveur répondait plus vite, assignait des adresses dans son propre sous-réseau, et les appareils — pensant avoir une IP valide — ne pouvaient plus joindre Internet.

Fix. Restreindre le serveur DHCP à l'interface réseau dédiée (`enp11s0`) et s'assurer que cette interface n'est accessible que depuis le switch isolé. La configuration systemd du service DHCP spécifie explicitement l'interface.

Leçon. En réseau, les services « ouverts » peuvent avoir des effets à large portée. Un serveur DHCP mal configuré est une attaque de type *rogue DHCP* involontaire. La portée des services réseau doit toujours être explicitement limitée.

5.3.3 Bug #3 — Les zéro frames enregistrées (le bug silencieux)

Symptôme

En développant le Mode Record, le CPU du Pi monte en flèche, mais 0 images sont reçues sur le serveur. Le flux MJPEG sur le port 8080 fonctionne parfaitement — la caméra filme bien.

Diagnostic. Dans `camera_manager.py`, la méthode `_reader_pipe()` lit les frames du flux caméra et met à jour deux variables : `_latest_jpeg` (pour le stream MJPEG) et `_latest_frames` (pour l'IA). La mise à jour de `_latest_jpeg` se faisait avant la tentative de décodage PIL. Mais la mise à jour du chronomètre `last_frame_time` (utilisé par le watchdog) se faisait après le bloc PIL.

La bibliothèque PIL *crashait silencieusement* sur le format JPEG généré par rpicam-vid — sans lever d'exception, sans message d'erreur. Le watchdog, ne voyant jamais `last_frame_time` mis à jour, croyait que la caméra était figée et retournait `None` à l'orchestrateur. L'orchestrateur n'avait donc rien à envoyer.

```
# AVANT (bug) :
self._latest_jpeg = frame_bytes
# ... PIL decode (peut crasher silencieusement)
self.last_frame_time = time.time()  # jamais atteint

# APRÈS (corrigé) :
self._latest_jpeg = frame_bytes
self.last_frame_time = time.time()  # mis à jour AVANT PIL
try:
    img = Image.open(io.BytesIO(frame_bytes))
    self._latest_frames = np.array(img)
except Exception:
    img = cv2.imdecode(np.frombuffer(frame_bytes, np.uint8),
                       cv2.IMREAD_COLOR)
    self._latest_frames = img
```

Listing 5.1. Correction du placement du chronomètre

Leçon. En programmation concurrente, deux branches de code peuvent traiter la même donnée avec des états divergents. Un bug « silencieux » (pas d'exception levée) est particulièrement difficile à détecter : tout semble fonctionner, mais rien ne se passe. Ce bug m'a coûté une journée et demie de diagnostic.

5.3.4 Bug #4 — Le scanner ARP sur une interface morte

Symptôme

Le script de Zero-Touch Provisioning network_scanner.py ne détecte aucun Raspberry Pi, même ceux qui sont clairement branchés sur le switch et accessibles en ping.

Diagnostic. Le service systemd configurait Scapy pour écouter sur l'interface `enx00051b82dc33` — un vieux dongle USB Ethernet qui traînait dans le tiroir et que j'avais branché lors d'un test. Ce dongle était physiquement débranché (état `NO-CARRIER`) au moment du test. Les Pi arrivaient sur l'interface de la carte mère, `enp11s0`, que Scapy n'écoutait pas.

Fix. Paramétrer explicitement le nom de l'interface dans le fichier de configuration du service systemd : `-interface enp11s0`.

Leçon. Les scripts réseau doivent toujours lier explicitement l'interface sur laquelle ils opèrent. Se fier à l'interface « par défaut » est une source d'ambiguïté qui se manifeste dans les pires moments — typiquement, le soir avant un tournoi.

5.3.5 Bug #5 — Le goulot d'étranglement N+1 de l'ORM Django

Symptôme

Le tableau de bord met plus de 5 secondes à charger lorsque plusieurs tables sont actives et que de nombreuses mains ont été jouées.

Diagnostic. Dans la vue `get_initial_data`, pour chaque table on cherchait la main la plus récente avec `t.hands.order_by('-created_at').first()`. Problème : en Django ORM, appeler `order_by()` sur un QuerySet qui a déjà été prefetché via `prefetch_related()` *annule le cache du prefetch* et génère une nouvelle requête SQL pour chaque appel. Résultat : une requête par table, une requête par main — c'est le fameux problème N+1.

Fix

```
# AVANT (N+1 queries) :
last_hand = table.hands.order_by('-created_at').first()

# APRÈS (utilise le prefetch, tri en Python) :
last_hand = sorted(
    table.hands.all(),
```

```
    key=lambda h: h.created_at,
    reverse=True
)[0] if table.hands.all() else None
```

Listing 5.2. Remplacement de order_by() par sorted() en Python

Leçon. Un ORM abstrait la base de données de façon élégante, mais il faut comprendre comment les requêtes SQL sous-jacentes sont construites. L'optimisation `prefetch_related()` peut être silencieusement annulée par une modification d'apparence anodine. L'outil de profiling Django (`django-debug-toolbar`) est indispensable pour détecter ce type de problème.

5.3.6 Bug #6 — Les connexions réseau fantômes

Symptôme

Après qu'un câble réseau est débranché d'un Pi, la table correspondante reste marquée « ACTIVE » sur le tableau de bord pendant plusieurs minutes, parfois indéfiniment.

Diagnostic. Une coupure réseau par débranchement de câble ne génère pas de message de fermeture TCP propre — la connexion disparaît physiquement sans que le serveur soit prévenu. Django Channels maintient alors le canal ouvert côté serveur, croyant que le Pi est toujours connecté.

Fix. Implémentation d'un watchdog dans `TableConsumer` :

```
async def watchdog(self):
    while True:
        await asyncio.sleep(60)
        elapsed = time.time() - self.last_stats_time
        if elapsed > 60:
            await self.close()  # Force la déconnexion
            await self.channel_layer.group_send(
                "dashboard",
                {"type": "table_status_update",
                 "table_id": self.table_id,
                 "is_active": False}
            )
            break
```

Listing 5.3. Watchdog Dead Man's Switch dans TableConsumer

Leçon. En réseau, l'absence de message n'est pas un acquittement. Un système qui dépend de notifications de déconnexion doit implémenter un mécanisme de liveness indépendant. Le pattern *Dead Man's Switch* — si je n'ai pas de nouvelles dans X secondes, je présume une défaillance — est la solution standard.

5.4 Métriques de performance

Table 5.1. Métriques observées en conditions terrain

Métrique	Valeur observée	Cible
Latence détection –> dashboard	200 – 400 ms	< 2 000 ms
FPS d'inférence (Hailo HAT)	15 – 20 fps	> 10 fps
FPS d'inférence (CPU fallback)	2 – 4 fps	–
Température Pi en charge	55 – 70°C	< 75°C
CPU Pi en charge	25 – 45%	< 80%
RAM Pi utilisée	1,2 – 1,8 Go	< 6 Go
Temps de provisioning ZTP	∼5 min	< 10 min

La latence de bout en bout (de la pose de la carte à son affichage sur le tableau de bord) est largement en dessous de la cible de 2 secondes. Le facteur limitant est la vitesse à laquelle l'arbitre humain joue ses cartes — typiquement toutes les 3 à 5 secondes — ce qui laisse une marge confortable.

5.5 Réflexion : déboguer un système distribué

Ce projet m'a appris quelque chose que les cours ne transmettent pas facilement : déboguer un système distribué est fondamentalement différent de déboguer un programme classique.

Quand un programme python plante, il y a une trace, une ligne, une erreur. Quand un système distribué se comporte mal, le symptôme visible peut être à l'opposé de la cause. Le « 0 frames » du bug #3 se manifestait côté serveur, mais la cause était dans le timing d'une variable côté Pi. Les connexions fantômes du bug #6 se manifestaient sur le tableau de bord, mais la cause était dans l'absence de mécanisme de liveness sur la connexion TCP.

La méthode que j'ai développée : instrumenter chaque composant avec des logs timestamps, observer le système pendant plusieurs minutes en conditions réelles, et suivre le fil des événements dans le temps. Les logs structurés — avec un timestamp précis, un identifiant de table, et un type d'événement — sont devenus mon outil principal de diagnostic.

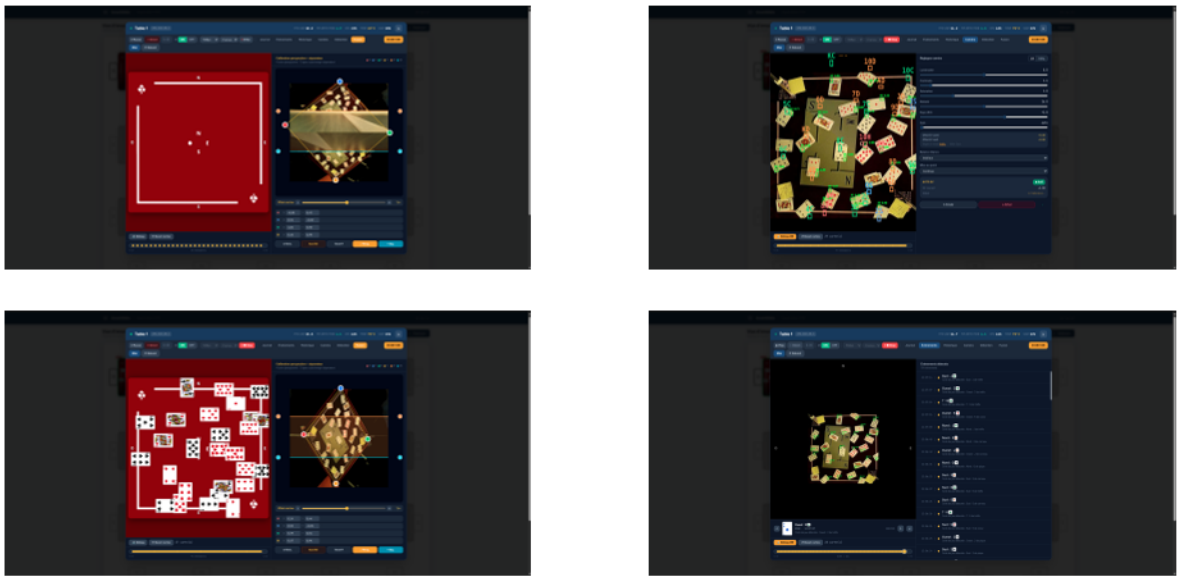


Figure 6.1. Galerie de l'interface utilisateur SmartKibitz — L'interface s'adapte aux besoins des arbitres (vue d'ensemble, flux direct, historique complet) pour une supervision exhaustive du tournoi.

Chapitre 6

Résultats et perspectives

6.1 État du système au terme du stage

6.1.1 Visualisation des résultats

L'interface finale offre une vision claire et variée de l'état des compétitions (voir Figure 6.1).

6.1.2 Ce qui fonctionne

Au 15 mai 2026, les fonctionnalités suivantes sont opérationnelles et validées en conditions terrain :

— **Pipeline IA complet** : caméra –> fusion –> Hailo AI HAT –> YOLOv8m –> détection des 52 cartes –> WebSocket –> base de données –> tableau de bord.
— **Streaming MJPEG** : aperçu vidéo en temps réel de chaque table, accessible depuis le navigateur.
— **Tableau de bord interactif** : grille 5 × 4, modale détaillée par table, statistiques système (CPU, RAM, température, FPS), journaux de démarrage.

— **Zero-Touch Provisioning** : validation sur plusieurs Pi, de la détection ARP au premier message WebSocket en moins de 5 minutes.
— **OTA Hot-Swap** : plusieurs mises à jour du code edge déployées en conditions réelles pendant le stage.
— **Résilience réseau** : watchdog serveur + backoff exponentiel client, testés par débranchement forcé de câbles.
— **Mode Record** : collecte de frames en conditions réelles pour le réentraînement du modèle.
— **Replay** : deux modes opérationnels — historique des coups par main (cards_log) et replay vidéo frame par frame avec timeline des détections et contrôle de vitesse.

6.1.3 Ce qui reste à faire

— **Déploiement sur 18 tables supplémentaires** : l'infrastructure est prête, le matériel n'est pas encore disponible en quantité.
— **Connexion du Hailo AI HAT** sur les deux Pi actuels (débranché pendant les tests de développement).
— **Réentraînement du modèle** sur les images collectées en Mode Record. Le dataset de la FFB est en cours de constitution.
— **Test de charge à 20 WebSockets simultanés** : les 2 Pi actuels ne permettent pas de simuler la charge complète.
— **Validation de la calibration triangulaire** avec les caméras en position définitive.

6.2 Perspectives d'évolution

6.2.1 Vers un scoring automatique

Le moteur de calcul de score a été développé en fin de stage dans `core/scoring.py`. La classe `BridgeScorer` calcule le score d'un contrat à partir du niveau, de la couleur, du nombre de levées réalisées, de la vulnérabilité et du type de contre. La fonction `get_trick_winner()` détermine le gagnant d'une levée à partir de la liste des cartes jouées et de l'atout. Ces deux briques permettent, en théorie, de reconstituer intégralement le résultat d'une main.

Ce qui manque encore, c'est le câblage entre ce moteur et le flux en temps réel : aujourd'hui, le score n'est pas affiché automatiquement sur le tableau de bord. L'intégration nécessite d'associer le log des cartes jouées à la distribution PBN importée, de déterminer le contrat annoncé (information que SmartKibitz ne capture pas encore — elle est saisie manuellement par l'opérateur de table dans le logiciel de scoring externe), et de déclencher le calcul à la 52^{e} carte. C'est la prochaine étape de développement, et les fondations sont en place.

6.2.2 Supervision multi-salles

Les grands championnats nationaux mobilisent plusieurs salles simultanément. Une évolution naturelle serait de permettre à plusieurs instances de SmartKibitz de se fédérer, avec un tableau de bord central agrégeant les données de plusieurs lieux.

Techniquement, cela nécessiterait de passer d'un système de communication en mémoire (Redis InMemoryChannelLayer) à une configuration Redis externe permettant la distribution entre plusieurs serveurs.

6.2.3 Aide à l'arbitrage par IA

La détection de renonces (jouer dans la mauvaise couleur) est aujourd'hui possible à partir du log des cartes et de la distribution PBN. Une alerte automatique pourrait être générée lorsqu'une carte jouée est incompatible avec les règles de suivi de couleur. Ce serait le premier système d'aide à l'arbitrage basé sur la vision par ordinateur dans le bridge compétitif francophone.

6.2.4 VAR bridge : la relecture vidéo

Ce n'est plus une perspective : c'est implémenté. Le replay vidéo frame par frame (décrit section 4.2.5) permet déjà à un arbitre de revoir les secondes qui précèdent une contestation. La prochaine étape serait d'indexer ces archives par table et par donne pour y accéder directement depuis le rapport de partie, sans avoir à naviguer manuellement dans la liste des sessions.

6.3 Impact sur la Fédération

Au-delà des fonctionnalités, ce projet a ouvert une réflexion interne à la FFB sur la numérisation de l'expérience de tournoi. La direction informatique a été enthousiaste à l'idée d'intégrer ces outils dans les prochains championnats officiels.

Il y a aussi un impact indirect : le Mode Record constitue la base d'un dataset de cartes de bridge en conditions réelles qui n'existait pas avant ce stage. Ce dataset a de la valeur au-delà du projet lui-même — il pourrait alimenter des travaux de recherche ou être partagé avec d'autres fédérations nationales de bridge.

Chapitre 7

Conclusion

7.1 Bilan professionnel

Revenons à la scène d'introduction. Cette fois, ajoutons un personnage que j'avais omis : l'opérateur de table, concentré sur son écran, qui saisit manuellement chaque carte jouée dans le logiciel de scoring. C'est lui le premier bénéficiaire de SmartKibitz. Pas l'arbitre — lui bénéficie du replay vidéo sur les contestations, ce qui est réel et utile, mais c'est un apport secondaire. L'impact principal est d'automatiser un travail de saisie répétitif, source d'erreurs, qui consomme l'attention de l'opérateur tout au long du tournoi.

Le système existe, il fonctionne, et sur les deux tables déployées il fait exactement ce qu'on lui demande. Ce n'est pas une vision : c'est un logiciel opérationnel, déployé sur du matériel réel.

Mais il serait malhonnête d'en rester là.

7.1.1 Ce que le système fait bien

L'architecture résiste. Les mécanismes de résilience — watchdog, backoff exponentiel, event sourcing — se sont comportés exactement comme prévu face aux coupures réseau, aux redémarrages, aux déboires matériels. Le Zero-Touch Provisioning fonctionne de manière répétable. L'OTA Hot-Swap a été utilisé de nombreuses fois pendant le stage sans incident.

La stack technique a des fondations solides : Django Channels avec Redis est une combinaison éprouvée en production ; PostgreSQL avec l'event sourcing offre une garantie de données sans faille ; le modèle YOLOv8m compilé pour Hailo atteint des performances d'inférence acceptables pour le cas d'usage.

7.1.2 Ce qui m'a résisté

La précision de détection sur les cartes à mi-distance, dans des conditions d'éclairage médiocres, reste un point de travail. Le modèle actuel — entraîné sur un dataset encore limité de parties réelles FFB — présente parfois des confusions entre cartes de valeurs proches (le 8 de Pique et le 8 de Trèfle, par exemple, ont des caractéristiques visuelles très similaires vues de dessus). C'est un problème de données autant que de modèle.

La calibration des caméras (mode triangulaire) nécessite une précision de montage

que les tables de tournoi ne permettent pas toujours. La fusion vstack reste le mode par défaut, et elle produit un résultat satisfaisant mais pas parfait.

7.2 Bilan personnel

Je savais coder avant ce stage. Je savais utiliser Python, manipuler des APIs, concevoir une base de données. Ce n'est pas ça que ce stage m'a appris.

7.2.1 Découvrir ce que je ne savais pas faire

La première surprise a été de réaliser à quel point un environnement professionnel est différent d'un projet académique. À l'IUT, les consignes sont claires, les délais connus à l'avance, les critères d'évaluation explicites. En stage, on navigue à vue. La demande initiale évolue, les contraintes se révèlent en chemin, les décisions ne sont jamais définitives.

Les premières semaines ont été déstabilisantes. Je ne savais pas comment organiser mes journées, comment communiquer l'avancement, comment poser une question sans avoir l'air d'ignorer quelque chose que je « devrais » savoir. Les daily meetings — ces courtes réunions quotidiennes de synchronisation — m'ont pris du temps à apprivoiser.

7.2.2 Découvrir ce que j'étais capable de faire

La deuxième surprise, plus agréable, a été de réaliser que j'étais capable de choses que je n'imaginais pas : concevoir une architecture réseau complète, rédiger un cahier des charges de boîtier pour l'impression 3D, structurer un plan de déploiement matériel à l'échelle d'une salle de tournoi. Des compétences loin du code, que personne ne m'avait enseignées explicitement.

Ce n'est pas que j'étais exceptionnellement doué pour ces tâches. C'est que la nécessité et la confiance accordée par Laurent et Zakaria m'ont conduit à les aborder sans filet. À l'IUT, on prépare des étudiants à être développeurs. Ce stage m'a montré que le métier de développeur, dans un contexte réel, déborde largement de cette définition.

7.2.3 La suite

Je cherche à rester à la FFB sous forme d'alternance, en parallèle d'un Master Ingénieur d'Affaires IATI. SmartKibitz n'est pas terminé. Les 18 tables restantes attendent, le modèle YOLOv8 s'améliore à chaque nouvelle donnée collectée, et le scoring automatique est à portée de quelques semaines de développement.

Ce projet a commencé dans un cours de Django à l'IUT d'Orsay, avec un professeur qui m'a fait confiance sur un projet universitaire ambitieux. Il continue dans les salles de tournoi de la Fédération Française de Bridge. La ligne droite entre les deux n'était pas évidente — c'est précisément ce qui la rend intéressante.

Bibliographie

[1] Django Software Foundation, "Django : The web framework for perfectionists with deadlines." https://www.djangoproject.com/, 2024. Version 5.2.

[2] A. Godwin and contributors, "Django channels — real-time django." https://channels.readthedocs.io/, 2024. Version 4.x.

[3] I. Fette and A. Melnikov, "The websocket protocol," Tech. Rep. RFC 6455, IETF, 2011.

[4] G. Jocher, A. Chaurasia, and J. Qiu, "Ultralytics yolov8." https://github.com/ultralytics/ultralytics, 2023. Version 8.

[5] Hailo Technologies, "Hailo-8 ai processor — technical overview." https://hailo.ai/products/ai-accelerators/hailo-8/, 2024. 13 TOPS edge AI accelerator.

[6] Raspberry Pi Foundation, "Raspberry pi 5 product brief." https://www.raspberrypi.com/products/raspberry-pi-5/, 2023.

[7] C. Porzio, "Alpine.js — a rugged, minimal framework for composing javascript behavior in your markup." https://alpinejs.dev/, 2024.

[8] Django Software Foundation, "Daphne — django asgi http/websocket server." https://github.com/django/daphne, 2024.

[9] M. Fowler, "Event sourcing," *martinfowler.com*, 2005.

[10] P. Biondi and contributors, "Scapy — packet crafting for python." https://scapy.net/, 2024.

[11] J. Forcier and contributors, "Paramiko — sshv2 protocol library for python." https://www.paramiko.org/, 2024.

[12] G. Bradski, "Opencv — open source computer vision library." https://opencv.org/, 2024.

[13] Raspberry Pi Foundation, "rpicam-apps — raspberry pi camera software." https://github.com/raspberrypi/rpicam-apps, 2024.

[14] Fédération Française de Bridge, "Fédération française de bridge — site officiel." https://www.ffbridge.fr/, 2024.

[15] S. Sanfilippo and contributors, "Redis — the open source, in-memory data store." https://redis.io/, 2024.

[16] H. van Staveren *et al.*, "Portable bridge notation specification." http://www.tistis.nl/pbn/, 1994.

Annexe A

Glossaire

Air-gap Réseau physiquement isolé, sans connexion à Internet ni à tout autre réseau extérieur.

Alpine.js Framework JavaScript minimaliste (15 Ko) permettant d'ajouter de la réactivité à une page HTML sans outillage de build.

ASGI *Asynchronous Server Gateway Interface*. Interface standard Python pour les serveurs web asynchrones, permettant les connexions longue durée (WebSocket, HTTP/2).

Backoff exponentiel Stratégie de reconnexion dans laquelle le délai entre deux tentatives double à chaque échec : 3s, 6s, 12s..., avec un plafond maximum. Prévient la surcharge du serveur en cas de redémarrage.

Bounding box Boîte englobante rectangulaire produite par un modèle YOLO pour localiser un objet détecté dans une image.

Consumer Dans Django Channels, équivalent asynchrone d'une vue Django pour les connexions WebSocket.

Daphne Serveur HTTP/WebSocket ASGI développé par l'équipe Django, utilisé pour servir les applications Django Channels.

Dead Man's Switch Mécanisme qui déclenche une action si aucun signal de vie n'est reçu dans un délai défini. Utilisé ici pour détecter les déconnexions réseau silencieuses.

DHCP *Dynamic Host Configuration Protocol*. Protocole réseau d'attribution automatique d'adresses IP.

Django Channels Extension de Django ajoutant le support des protocoles asynchrones (WebSocket, SSE) à travers une couche de messagerie (channel layer).

Donne Distribution précise des 52 cartes entre les 4 joueurs d'une partie de bridge. Définie par un format PBN standardisé.

Edge AI Intelligence artificielle embarquée directement sur les appareils en périphérie du réseau (Pi, téléphones, caméras), par opposition à une IA centralisée sur un serveur cloud.

EdgeDeployment
: Modèle Django représentant une version du code déployable sur les nœuds edge, avec son manifeste de fichiers et de checksums SHA256.

Event Sourcing
: Patron de conception dans lequel l'état du système est représenté par un journal d'événements immuable, plutôt que par un état courant mutable.

Hailo AI HAT
: Accélérateur neuronal de Hailo Technologies, montable sur le Raspberry Pi 5 en tant que HAT (*Hardware Attached on Top*). Offre 26 TOPS pour une consommation de 5 W.

HEF
: *Hailo Executable Format*. Format binaire produit par le compilateur Hailo, contenant le graphe de calcul du modèle optimisé pour le silicium Hailo-8.

HMAC-SHA256
: Algorithme d'authentification de message basé sur une fonction de hachage (SHA-256) et une clé secrète partagée.

IMX708
: Capteur CMOS Sony équipant le Camera Module 3 de Raspberry Pi. Résolution native 12 Mpx (2304 × 1296 px en mode haute résolution).

Levée
: Unité de jeu au bridge : chacun des 4 joueurs pose une carte, le joueur qui a posé la carte la plus forte de la couleur demandée remporte la levée.

MJPEG
: *Motion JPEG*. Format de streaming vidéo dans lequel chaque frame est envoyée individuellement sous forme de JPEG, sans compression inter-frames.

Nœud edge
: Dans SmartKibitz, désigne l'ensemble Raspberry Pi + Hailo AI HAT + caméras installé sur chaque table.

N+1 queries
: Problème de performance ORM dans lequel une requête principale (N) génère N requêtes supplémentaires (une par élément de résultat) au lieu d'utiliser une jointure ou un prefetch.

OTA
: *Over-The-Air*. Mécanisme de mise à jour logicielle à distance, sans accès physique au matériel.

PBN
: *Portable Bridge Notation*. Format texte standard international pour représenter les donnes, enchères et résultats de bridge.

PoE
: *Power over Ethernet*. Standard permettant de transmettre l'alimentation électrique et les données réseau sur le même câble RJ45.

Prefetch_related
: Méthode ORM Django permettant de précharger en mémoire les relations entre objets en une seule requête SQL, pour éviter les N+1

queries.

Provisionnement
: Processus d'installation et de configuration initiale d'un appareil (ici, un Raspberry Pi) pour l'intégrer dans un système.

Redis
: Système de stockage de données en mémoire, utilisé ici comme *channel layer* par Django Channels pour la diffusion de messages entre consumers.

Renonce
: Irrégularité au bridge consistant à jouer une carte d'une couleur différente de celle demandée alors qu'on en possède.

Symlink atomique
: Remplacement d'un lien symbolique par un autre en une seule opération système (`os.rename()`), garantissant qu'il n'y a jamais d'état intermédiaire.

Thundering Herd
: Phénomène dans lequel un grand nombre de clients tentent simultanément de se reconnecter à un serveur après sa remise en service, générant une surcharge.

TOPS
: *Tera Operations Per Second*. Unité de mesure de la puissance de calcul des accélérateurs IA.

UDP Broadcast
: Envoi d'un datagramme UDP à toutes les machines d'un réseau local simultanément.

WSGI
: *Web Server Gateway Interface*. Interface synchrone standard pour les serveurs web Python, incompatible avec les connexions WebSocket longue durée.

YOLOv8
: *You Only Look Once v8*. Famille de modèles de détection d'objets en temps réel développée par Ultralytics. Utilisé ici dans sa variante *medium* (YOLOv8m) pour détecter les 52 types de cartes.

Zero-Touch Provisioning (ZTP)
: Déploiement et configuration automatiques d'un appareil réseau sans intervention manuelle sur le matériel.

Annexe B

Arborescence du projet

```
bridgeDash/
+-- manage.py                       # Point d'entrée Django
+-- requirements.txt                # Dépendances Python serveur
+-- .env                            # Configuration (non versionné)
+-- bridge_server/                  # Configuration Django
|   +-- settings.py
|   +-- asgi.py                     # Configuration Daphne ASGI
|   +-- urls.py
+-- core/                           # Application Django principale
|   +-- models.py                   # Table, Board, Hand, AuditLog...
|   +-- consumers.py                # TableConsumer + DashboardConsumer
|   +-- views.py                    # Endpoints HTTP (lean)
|   +-- services.py                 # Helpers partagés
|   +-- inference.py                # Inférence GPU serveur (fallback)
|   +-- recording.py                # Mode Record
|   +-- routing.py                  # Routage WebSocket
|   +-- migrations/                 # 13 migrations de schema
+-- edge_node/                      # Code Raspberry Pi
|   +-- edge_client.py              # Orchestrateur asyncio
|   +-- camera_manager.py           # Gestion dual-caméra
|   +-- yolo_inference.py           # Inférence Hailo HAT
|   +-- warping_utils.py            # Fusion d'images
|   +-- stream_server.py            # Serveur MJPEG port 8080
|   +-- bootstrap.py                # Agent OTA (symlink atomique)
|   +-- network_config.py           # Découverte UDP
|   +-- setup.sh                    # Script de provisionnement
|   +-- bridge_edge.service         # Unité systemd
+-- scripts/
|   +-- auto_provision.py           # Provisioner SSH/SCP
|   +-- network_scanner.py          # Scanner ARP (Scapy)
+-- templates/
|   +-- dashboard.html              # Tableau de bord (grille + modale)
|   +-- base.html                   # Template de base
+-- static/
|   +-- js/
|       +-- dashboard_app.js        # Application Alpine.js
|       +-- alpine.min.js           # Alpine.js local
|       +-- tailwind.js             # Tailwind CSS local
+-- docs/                           # Documentation technique complète
+-- systemd/                        # Fichiers de service serveur
```

Listing B.1. Structure des répertoires du projet SmartKibitz

Annexe C

Référence des endpoints API

Table C.1. Endpoints HTTP (core/urls.py)

Route	Méthode	Authentification	Fonction
`/`	GET	Non	Tableau de bord principal
`/api/register_device/`	POST	Non	Assignation table par MAC
`/api/pi_control/`	POST	API Key	Reboot / Shutdown
`/api/record_frame/`	POST	Non	Réception frame JPEG
`/api/inference/`	POST	Non	Inférence GPU serveur
`/api/config/`	GET	Non	Config inférence pour Pi
`/ws/table/{id}/`	WS	Non	Canal Pi ↔ Serveur
`/ws/dashboard/`	WS	Non	Canal Serveur ↔ Navigateur

Annexe D

Messages WebSocket complets

Table D.1. Messages Pi –> Serveur

Type	Champs principaux	Fréquence
`card_event`	table_id, cards[], board_id	1-5 fps
`stats`	cpu, ram, temp, fps, uptime	1 Hz
`alert`	severity, message	Sur événement
`sync`	local_log[], timestamp	Reconnexion
`setup_log`	step, message	Boot
`camera_settings_ack`	success, new_settings	Sur commande

Table D.2. Messages Serveur –> Dashboard

Type	Champs principaux	Déclencheur
`initial_state`	tables[], boards[]	Connexion navigateur
`dashboard_update`	table_id, new_cards[]	Nouvelles cartes
`table_status_update`	table_id, is_active	Pi connect/disconnect
`table_alert`	table_id, severity, message	Pi envoie alert
`table_stats_update`	table_id, cpu, ram, temp, fps	Pi envoie stats
`table_setup_log`	table_id, step, message	Pi envoie setup_log