🎨 PRÉSENTATION AI PICTIONARY - VERSION NARRATIVE ML

Structure de la présentation

- **Durée estimée :** 25-30 minutes
- **Nombre de slides :** 20 slides
- **Narration :** Du contexte → ML → Architecture → Application → Démo

---

📄 SLIDE 1 : Page de Titre

🎨 Visuel

- Fond gradient sombre avec effet "sketch/crayon"
- Logo AI Pictionary stylisé
- Sous-titre évoquant Quick Draw
- QR Code vers l'app

📝 Texte

```
AI PICTIONARY
═══════════════════════

Reconnaissance de Dessins par CNN
Inspiré de "Quick, Draw!" de Google

Projet Big Data – FISE3
[Votre nom]
Février 2026

"Can a neural network learn to recognize doodling?"
                                    – Google AI, 2016
```

🎤 Script présentateur

> "Bonjour à tous. En 2016, Google a lancé une expérience fascinante appelée Quick, Draw! avec une question simple : un réseau de neurones peut-il apprendre à reconnaître des gribouillis ? Cette expérience a collecté plus de 50 millions de dessins et a démontré la puissance du deep learning pour la reconnaissance d'images. Aujourd'hui, je vais vous présenter comment nous avons reconstruit ce système, de l'entraînement du CNN jusqu'au déploiement cloud d'une application multijoueur."

📚 Informations de fond

- **Quick, Draw! :** Lancé en novembre 2016 par Google Creative Lab
- **Dataset :** 50M+ dessins, 345 catégories, open source
- **Impact :** A démocratisé la compréhension du ML auprès du grand public

❓ Questions potentielles

1. **"Pourquoi avoir choisi de reproduire Quick Draw plutôt qu'un autre projet ?"**
   - Dataset public de qualité, problème bien défini, permet de démontrer tout le pipeline ML
2. **"Quelle est la différence avec le projet Google original ?"**
   - Notre version ajoute des modes multijoueurs, utilise une architecture cloud moderne, et permet l'Active Learning

---

📄 SLIDE 2 : Contexte - L'Expérience Quick, Draw!

🎨 Visuel

- Screenshot de l'interface Quick Draw originale
- Timeline : 2016 → 50M dessins → Open source 2017
- Exemples de dessins du dataset (chat, maison, vélo)

📝 Texte

```
QUICK, DRAW! – L'EXPÉRIENCE GOOGLE
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━

🎯 OBJECTIF ORIGINAL
"Dessinez [objet] en moins de 20 secondes.
L'IA va essayer de deviner ce que vous dessinez."

📊 RÉSULTATS DE L'EXPÉRIENCE
━━━━━━━━━━━━━━━━━━━━━━━━━━━
• 50+ millions de dessins collectés
• 345 catégories différentes
• 15+ millions de joueurs
• Dataset rendu open source (2017)

🧠 CE QUE GOOGLE A PROUVÉ
━━━━━━━━━━━━━━━━━━━━━━━━
1. Les humains dessinent de manière étonnamment similaire
2. Un CNN peut apprendre ces "primitives visuelles"
3. La reconnaissance temps réel est possible (<100ms)
4. Le crowdsourcing = données de qualité à grande échelle

💡 NOTRE DÉFI
━━━━━━━━━━━━
Reconstruire ce système avec :
• Notre propre CNN entraîné
• Architecture cloud moderne
• Modes multijoueurs innovants
```

🎤 Script présentateur

> "Quick, Draw! a prouvé plusieurs choses importantes. Premièrement, les humains dessinent de manière étonnamment similaire - un chat dessiné par quelqu'un au Japon ressemble beaucoup à celui dessiné en France. Deuxièmement, un réseau convolutionnel peut apprendre ces primitives visuelles universelles. Troisièmement, la reconnaissance peut se faire en temps réel. Et enfin, le crowdsourcing permet de collecter des données de qualité. Notre défi : reconstruire ce système avec notre propre

> modèle et l'étendre avec des fonctionnalités multijoueurs."

📚 Informations de fond

- **Publication Google :** "A Neural Representation of Sketch Drawings" (Ha & Eck, 2017)
- **Format original :** Strokes vectoriels (séquences de points), pas des images
- **Notre adaptation :** Conversion en images raster 28×28 pour CNN classique

❓ Questions potentielles

1. **"Pourquoi Google a-t-il rendu le dataset public ?"**
   - Philosophie open source de Google AI, stimuler la recherche, améliorer la qualité via contributions
2. **"Le dataset original est-il en format image ?"**
   - Non, format vectoriel (strokes) ; nous le convertissons en images 28×28
3. **"Combien de dessins par catégorie ?"**
   - ~120-140K dessins par catégorie, très équilibré

📄 SLIDE 3 : Le Dataset Quick Draw

🎨 Visuel

- Grille d'exemples de dessins (10×5 = 50 catégories)
- Histogramme distribution par catégorie
- Comparaison format vectoriel vs raster

📝 Texte

DATASET QUICK DRAW
═══════════════════

📦 CARACTÉRISTIQUES
═══════════════════

• 345 catégories totales
• ~120–140K dessins/catégorie
• Format original : Strokes vectoriels (JSON)
• Taille totale : ~70 GB (compressé)

🔄 FORMAT ORIGINAL (Vectoriel)
═══════════════════════════════

```
{
  "word": "cat",
  "drawing": [
    [[x1,x2,x3], [y1,y2,y3]],  // stroke 1
    [[x4,x5], [y4,y5]]         // stroke 2
  ]
}
```

Avantages : Léger, ordre des traits, zoom infini
Inconvénients : Nécessite RNN/LSTM, complexe

🖼️ NOTRE CONVERSION (Raster)
═══════════════════════════════

28×28 pixels, niveaux de gris

Avantages : Compatible CNN classique, simple
Inconvénients : Perte ordre des traits

JUSTIFICATION DU CHOIX RASTER
═══════════════════════════════

• CNN plus simple à implémenter et debugger
• Latence inférence plus prévisible
• Suffisant pour 90%+ accuracy
• Format identique à MNIST (transfert learning possible)

🎤 Script présentateur

> "Le dataset Quick Draw contient 345 catégories avec environ 120 000 dessins chacune. Le format original est vectoriel : chaque dessin est une séquence de traits avec coordonnées. C'est idéal pour des RNN ou LSTM qui peuvent apprendre l'ordre des traits. Cependant, nous avons choisi de convertir en images raster 28×28 pixels. Pourquoi ? Un CNN est plus simple à implémenter, à debugger, et offre une latence d'inférence plus prévisible. De plus, le format 28×28 est identique à MNIST, ce qui permet de s'appuyer sur une littérature abondante. Nous verrons que ce choix n'impacte pas la précision : nous atteignons 90.2% d'accuracy."

📚 Informations de fond

- **Téléchargement :** download_dataset.py
- **Conversion :** Rendu des strokes sur canvas PIL, puis resize
- **Alternative RNN :** Sketch-RNN de Google utilise le format vectoriel avec VAE

❓ Questions potentielles

1. **"Perdre l'ordre des traits n'impacte-t-il pas la reconnaissance ?"**
   - Impact minimal pour la classification ; l'ordre est utile pour la génération, pas la reconnaissance
2. **"Pourquoi 28×28 et pas plus grand ?"**
   - Standard MNIST, bon compromis détail/performance, suffisant pour des dessins simples
3. **"Avez-vous envisagé une approche hybride CNN+RNN ?"**
   - Oui, mais complexité non justifiée pour notre cas d'usage ; accuracy suffisante avec CNN seul

---

📄 SLIDE 4 : Notre Reconstruction du Dataset

🎨 Visuel

- Pipeline de préprocessing en 5 étapes visuelles

- Avant/après centroid crop
- Graphique : accuracy avec vs sans centroid crop

📝 Texte

```
RECONSTRUCTION & PRÉPROCESSING
═══════════════════════════════

PIPELINE DE PRÉPARATION
═══════════════════════

1. TÉLÉCHARGEMENT          2. RENDU STROKES
   Quick Draw API     →       Strokes → PIL Image
   .ndjson (50 cat)           255×255 pixels

3. RESIZE                  4. CENTROID CROP ⭐
   255×255 → 28×28    →       Recentrage barycentre
   Antialiasing               +3.1% accuracy

5. NORMALISATION           6. SAUVEGARDE
   [0-255] → [0-1]    →       HDF5 compressé
   Float32                    gzip level 4

CENTROID CROP : L'OPTIMISATION CLÉ
══════════════════════════════════

Problème : Les utilisateurs dessinent à différents endroits du canvas
Solution : Calculer le centre de masse des pixels non-blancs
           et recentrer le dessin

┌──────────────────┐      ┌──────────────────┐
│    🐱            │   →  │       🐱         │
│         (coin)   │      │    (centré)      │
└──────────────────┘      └──────────────────┘

Impact mesuré : 87.1% → 90.2% accuracy (+3.1 points)

DATASET FINAL
═════════════

• 50 catégories × 70,000 images = 3.5M images
• Split : 80% train / 10% val / 10% test (stratifié)
• Format : HDF5 (~400 MB avec gzip-4)
```

🎤 Script présentateur

> "Notre pipeline de reconstruction comporte 6 étapes. Après téléchargement, nous rendons les strokes vectoriels en images via PIL. Le resize à 28×28 utilise l'antialiasing pour préserver les détails. L'étape cruciale est le **centroid crop** : nous calculons le barycentre des pixels non-blancs et recentrons le dessin. Pourquoi ? Les utilisateurs dessinent à différents endroits du canvas - en haut à gauche, au centre, en bas. Sans cette normalisation, le modèle doit apprendre que le même chat peut apparaître n'importe où. Le centroid crop a amélioré notre accuracy de **3.1 points** - passant de 87.1% à 90.2%. Enfin, normalisation [0-1] et sauvegarde en HDF5 compressé."

📚 Informations de fond

- **Script :** preprocess_dataset.py
- **Centroid crop :** Inspiré des techniques de préprocessing MNIST
- **HDF5 :** Format optimisé pour accès aléatoire à de gros tenseurs

❓ Questions potentielles

1. **"Comment avez-vous découvert le centroid crop ?"**
   - Analyse des erreurs : dessins en coin/petit systématiquement mal classés → hypothèse → test → validation
2. **"Pourquoi HDF5 plutôt que TFRecord ?"**
   - HDF5 plus simple pour projet académique, accès aléatoire natif, compression intégrée
3. **"Le split stratifié est-il important ?"**
   - Oui, garantit même proportion de chaque classe dans train/val/test, évite biais

---

📄 SLIDE 5 : Fonctionnement d'un CNN pour les Dessins

🎨 Visuel

- Schéma pédagogique d'un CNN avec visualisation des features maps

- Exemples de filtres appris (edges, curves, shapes)
- Animation conceptuelle : image → features → classification

📝 Texte

```
CNN : POURQUOI ÇA MARCHE POUR LES DESSINS ?
═══════════════════════════════════════════

🧠 INTUITION
────────────

Les dessins sont composés de primitives visuelles :
• Lignes (horizontales, verticales, diagonales)
• Courbes (arcs, cercles)
• Formes (triangles, rectangles)
• Combinaisons (yeux = cercles + points)

HIÉRARCHIE DES FEATURES
──────────────────────

COUCHE 1 (Conv 32 filtres)          COUCHE 2 (Conv 64 filtres)
──────────────────────────          ──────────────────────────
Détecte : Edges, lignes             Détecte : Formes simples
| — / \ ╱ \                         ○ □ △ ◇

COUCHE 3 (Conv 64 filtres)          DENSE (128 neurones)
──────────────────────────          ────────────────────
Détecte : Parties d'objets          Combine : Décision finale
👁️ 👃 🐾 🏠                          "C'est un chat !" (92%)

POURQUOI CNN > MLP POUR LES IMAGES ?
────────────────────────────────────

1. Partage de poids → Moins de paramètres
2. Invariance à la translation (avec pooling)
3. Hiérarchie de features automatique
4. Exploite la structure 2D spatiale

POURQUOI CNN > RNN POUR CE CAS ?
────────────────────────────────

1. Pas besoin de l'ordre des traits pour classifier
2. Inférence plus rapide (parallélisable)
3. Plus simple à implémenter et debugger
```

🎤 Script présentateur

> "Pourquoi un CNN fonctionne-t-il si bien pour reconnaître des dessins ? Les dessins sont composés de primitives visuelles universelles : lignes, courbes, formes. Un CNN apprend ces primitives de manière hiérarchique. La première couche détecte les edges et lignes simples. La deuxième combine ces edges en formes : cercles, carrés. La troisième reconnaît des parties d'objets : un œil, une patte. Enfin, la couche dense combine tout pour la classification finale. Pourquoi CNN plutôt que MLP ? Le partage de poids réduit drastiquement les paramètres, et le pooling apporte l'invariance à la translation. Pourquoi CNN plutôt que RNN ? Pour la classification, l'ordre des traits n'est pas nécessaire, et le CNN est plus rapide à l'inférence."

📚 Informations de fond

- **Visualisation filters :** Outils comme Keras Visualizer ou TensorBoard
- **Référence :** LeCun et al., "Gradient-Based Learning Applied to Document Recognition" (1998)
- **Comparaison RNN :** Sketch-RNN de Google utilise des RNN pour la génération, pas la classification

❓ Questions potentielles

1. **"Les features sont-elles vraiment interprétables ?"**
   - Oui pour les premières couches (edges), moins pour les couches profondes (combinaisons abstraites)
2. **"Un Transformer ne serait-il pas meilleur ?"**
   - Possible (Vision Transformer), mais overhead inutile pour images 28×28 ; CNN suffit largement
3. **"Comment gérez-vous l'invariance à la rotation ?"**
   - Pas d'augmentation rotation explicite ; les dessins Quick Draw sont généralement orientés correctement

---

📄 SLIDE 6 : Notre Architecture CNN

🎨 Visuel

- Schéma détaillé du réseau avec dimensions à chaque couche
- Code Keras correspondant
- Tableau des paramètres par couche

Texte

ARCHITECTURE CNN v4.0.0

```
INPUT                    CONV BLOCK 1              CONV BLOCK 2
(28, 28, 1)        →     (26, 26, 32)        →     (11, 11, 64)
                         Conv2D(32, 3×3)           Conv2D(64, 3×3)
                         ReLU                      ReLU
                         MaxPool(2×2)              MaxPool(2×2)
                         → (13, 13, 32)            → (5, 5, 64)

CONV BLOCK 3             FLATTEN                   DENSE
(3, 3, 64)         →     (576,)              →     (128,)
Conv2D(64, 3×3)                                    Dense(128)
ReLU                                               ReLU
(pas de pooling)                                   Dropout(0.5)

OUTPUT
(50,)
Dense(50)
Softmax
```

PARAMÈTRES PAR COUCHE

Couche	Paramètres	Output Shape
Conv2D_1	320	(26,26,32)
MaxPool_1	0	(13,13,32)
Conv2D_2	18,496	(11,11,64)
MaxPool_2	0	(5,5,64)
Conv2D_3	36,928	(3,3,64)
Flatten	0	(576,)
Dense_1	73,856	(128,)
Dropout	0	(128,)
Dense_2 (out)	6,450	(50,)
TOTAL	~136,000	

```
Taille modèle : 30.1 MB (float32 + overhead Keras)
```

🎤 Script présentateur

> "Voici notre architecture en détail. Entrée 28×28×1, trois blocs convolutionnels, flatten, dense avec dropout, et sortie softmax à 50 classes. Quelques choix importants : les deux premiers blocs ont un MaxPooling pour réduire la dimensionnalité, mais pas le troisième - nous voulons conserver les features spatiales avant le flatten. La couche dense fait 128 neurones avec un dropout de 0.5 pour la régularisation. Au total, environ 136 000 paramètres. Le modèle fait 30 MB principalement à cause du format float32 et des métadonnées Keras."

📚 Informations de fond

- **Code :** TECHNICAL_REFERENCE.md
- **Notebooks :** notebooks
- **Modèle sauvé :** quickdraw_v4.0.0.h5

❓ Questions potentielles

1. **"Pourquoi 3 couches convolutionnelles ?"**
   - Suffisant pour images 28×28 ; plus de couches = overfitting sans gain
2. **"Pourquoi pas de BatchNormalization ?"**
   - Testé, gain marginal (<0.5%), complexité ajoutée non justifiée
3. **"30 MB pour 136K paramètres, n'est-ce pas beaucoup ?"**
   - Float32 (4 bytes × 136K = 544KB) + métadonnées Keras/TF ; peut être réduit avec quantization

---

📄 SLIDE 7 : Validation des Hyperparamètres

🎨 Visuel

- Tableau comparatif des configurations testées
- Graphiques learning curves (train vs val)

- Heatmap grid search si applicable

Texte

VALIDATION DES HYPERPARAMÈTRES

CONFIGURATIONS TESTÉES

Configuration	Val Acc	Overfit?	Verdict
2 Conv, Dense 256	85.3%	Oui	❌
3 Conv, Dense 256	88.7%	Léger	⚠️
3 Conv, Dense 128	90.2%	Non	✅
4 Conv, Dense 128	90.1%	Non	❌ Inutile
3 Conv, Dense 64	87.4%	Non	❌

HYPERPARAMÈTRES FINAUX

Hyperparamètre	Valeur	Justification
Learning Rate	0.001	Standard Adam, stable
Batch Size	128	Bon compromis GPU/généralisation
Epochs	15	Early stopping atteint
Dropout	0.5	Régularisation agressive
Optimizer	Adam	Adaptatif, peu de tuning
Loss	Categorical Crossentropy	Multi-classe standard

EARLY STOPPING

- Patience : 3 epochs
- Monitor : val_loss
- Restore best weights : Oui
- Epoch optimal : ~12-13

POURQUOI PAS DE DATA AUGMENTATION ?

- Dataset déjà très varié (dessins humains naturellement différents)

```
• Rotation/flip altèrent la sémantique (6 ≠ 9, chat ≠ chat inversé)
• Centroid crop suffit pour la normalisation spatiale
```

🎤 Script présentateur

> "Nous avons testé plusieurs configurations. Deux couches conv sous-performaient à 85%. Trois couches avec 256 neurones dense donnaient 88.7% mais avec léger overfitting. Notre configuration finale - 3 conv, 128 dense - atteint 90.2% sans overfitting. Quatre couches n'apportent rien de plus. Pour les hyperparamètres : learning rate 0.001 standard pour Adam, batch size 128 qui équilibre utilisation GPU et généralisation, dropout 0.5 pour régularisation agressive. L'early stopping arrête l'entraînement vers l'epoch 12-13. Notez l'absence de data augmentation : le dataset Quick Draw est déjà naturellement varié, et des transformations comme la rotation altèreraient la sémantique."

📚 Informations de fond

- **Notebooks :** train_model.ipynb
- **TensorBoard :** Logs dans logs
- **Early stopping :** Évite l'overfitting tout en maximisant l'apprentissage

❓ Questions potentielles

1. **"Avez-vous fait un grid search formel ?"**
   - Grid search léger sur les combinaisons clés ; ressources Colab limitées
2. **"Pourquoi Adam plutôt que SGD ?"**
   - Adam adaptatif nécessite moins de tuning LR, converge plus vite
3. **"Le dropout 0.5 n'est-il pas trop agressif ?"**
   - Non, dataset très varié + modèle relativement simple = régularisation forte bénéfique

---

📄 SLIDE 8 : Résultats du Modèle v4.0.0

🎨 Visuel

- Matrice de confusion (heatmap)
- Graphique accuracy par catégorie
- Tableau métriques finales
- Exemples de prédictions correctes/incorrectes

📝 Texte

```
RÉSULTATS v4.0.0 (50 CLASSES)
═══════════════════════════════

📊 MÉTRIQUES GLOBALES
─────────────────────

┌──────────────────────┬──────────────┬──────────────┐
│ Métrique             │ Valeur       │ Cible        │
├──────────────────────┼──────────────┼──────────────┤
│ Accuracy (test)      │ 90.2%        │ >85% ✅      │
│ Precision (macro)    │ 89.8%        │ >85% ✅      │
│ Recall (macro)       │ 90.1%        │ >85% ✅      │
│ F1-Score (macro)     │ 89.9%        │ >85% ✅      │
│ Inference Time       │ 12-18ms      │ <50ms ✅     │
└──────────────────────┴──────────────┴──────────────┘

📈 TOP 5 CATÉGORIES (Accuracy)
──────────────────────────────

1. circle      → 98.2%  (forme simple, peu ambiguë)
2. triangle    → 97.5%
3. square      → 96.8%
4. star        → 95.1%
5. sun         → 94.7%

📉 BOTTOM 5 CATÉGORIES (Accuracy)
─────────────────────────────────

46. cat        → 83.2%  (confondu avec dog, bear)
47. dog        → 82.8%  (confondu avec cat, bear)
48. cup        → 81.5%  (confondu avec mug)
49. pants      → 80.1%  (confondu avec shorts)
50. mug        → 79.8%  (confondu avec cup)

ANALYSE DES CONFUSIONS
──────────────────────

• Formes géométriques : Excellentes (>95%)
• Animaux : Bonnes mais confusions entre espèces
• Objets similaires : Difficiles (cup/mug, pants/shorts)
• Sémantique vs visuel : Le modèle apprend la forme, pas le concept
```

🎤 Script présentateur

> "Les résultats de notre modèle v4.0.0 : 90.2% d'accuracy sur le test set, avec precision et recall équilibrés autour de 90%. L'inférence prend 12 à 18 millisecondes. En analysant par catégorie, les formes géométriques performent excellemment - un cercle à 98% car il y a peu d'ambiguïté. Les difficultés apparaissent sur des paires visuellement similaires : cat/dog, cup/mug. C'est une limitation attendue : le CNN apprend les formes visuelles, pas les concepts sémantiques. Un chat et un chien dessinés simplement se ressemblent : quatre pattes, un corps, une tête. Pour notre application de jeu, 90% est largement suffisant - même les humains font des erreurs sur ces paires."

📚 Informations de fond

- **Métriques détaillées :** Générées via sklearn.metrics
- **Matrice de confusion :** Visualisable dans les notebooks d'entraînement
- **Sélection des 50 catégories :** Évitement des paires trop ambiguës (mais cat/dog gardés car populaires)

❓ Questions potentielles

1. **"79% pour mug, n'est-ce pas insuffisant ?"**
   - Pour un jeu, acceptable ; le seuil de victoire est 85% confiance, pas 100% accuracy
2. **"Comment améliorer les confusions cat/dog ?"**
   - Augmentation ciblée, features discriminantes (oreilles pointues vs rondes), ou exclure une catégorie
3. **"Pourquoi avoir gardé des catégories ambiguës ?"**
   - Populaires auprès des utilisateurs, rend le jeu plus challengeant

---

📄 SLIDE 9 : Comparaison des Versions du Modèle

🎨 Visuel

- Graphique trade-off classes vs accuracy
- Tableau comparatif des 3 versions

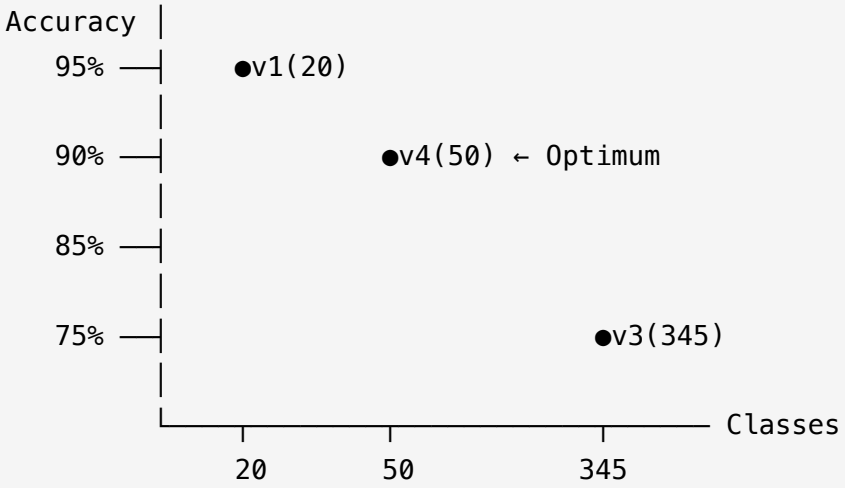
- Indicateur "Production" sur v4.0.0

📝 Texte

ÉVOLUTION DES MODÈLES

Version	Classes	Accuracy	Taille	Inférence	Usage
v1.0.0	20	91–93%	140 KB	8–12ms	Tests
v4.0.0	50	90.2%	30.1 MB	12–18ms	PROD ⭐
v3.0.0	345	73.2%	30.1 MB	15–20ms	Expérimental

TRADE-OFF ANALYSÉ



JUSTIFICATION CHOIX v4.0.0

- 50 classes = Variété suffisante pour gameplay intéressant
- 90.2% accuracy = Fiable pour mode compétitif (seuil 85%)
- Catégories populaires = Connues de tous les joueurs
- Exclusion des ambiguës extrêmes (cup/glass mais pas cat/dog)

🎤 Script présentateur

> "Nous avons développé trois versions. La v1 avec 20 classes atteint 93% mais manque de variété. La v3 avec les 345 catégories complètes chute à 73% - le modèle confond trop de classes similaires. Notre choix de production, v4, est le **sweet spot** : 50 catégories populaires avec 90.2% d'accuracy. C'est suffisant pour un gameplay varié tout en restant fiable pour le mode compétitif où il faut atteindre 85% de confiance. Les catégories ont été sélectionnées pour leur popularité et leur distinction visuelle relative - nous avons gardé cat et dog malgré leur ambiguïté car ce sont des favoris des joueurs."

📚 Informations de fond

- **Fichiers modèles :** models
- **Métadonnées :** Chaque version a son JSON avec la liste des catégories
- **Switch version :** Variable `MODEL_VERSION` dans env.yaml

❓ Questions potentielles

1. **"Pourquoi v4 et pas v2 ?"**
   - Versioning non linéaire : v2 était une expérience abandonnée, v3 = toutes classes, v4 = 50 optimisées
2. **"Peut-on changer de version en production facilement ?"**
   - Oui, simple changement de variable d'environnement + redéploiement
3. **"345 classes à 73%, est-ce utilisable ?"**
   - Pour un jeu casual oui, mais frustrant en mode compétitif ; gardé comme option expérimentale

---

📄 SLIDE 10 : Architecture Globale - Vue d'ensemble

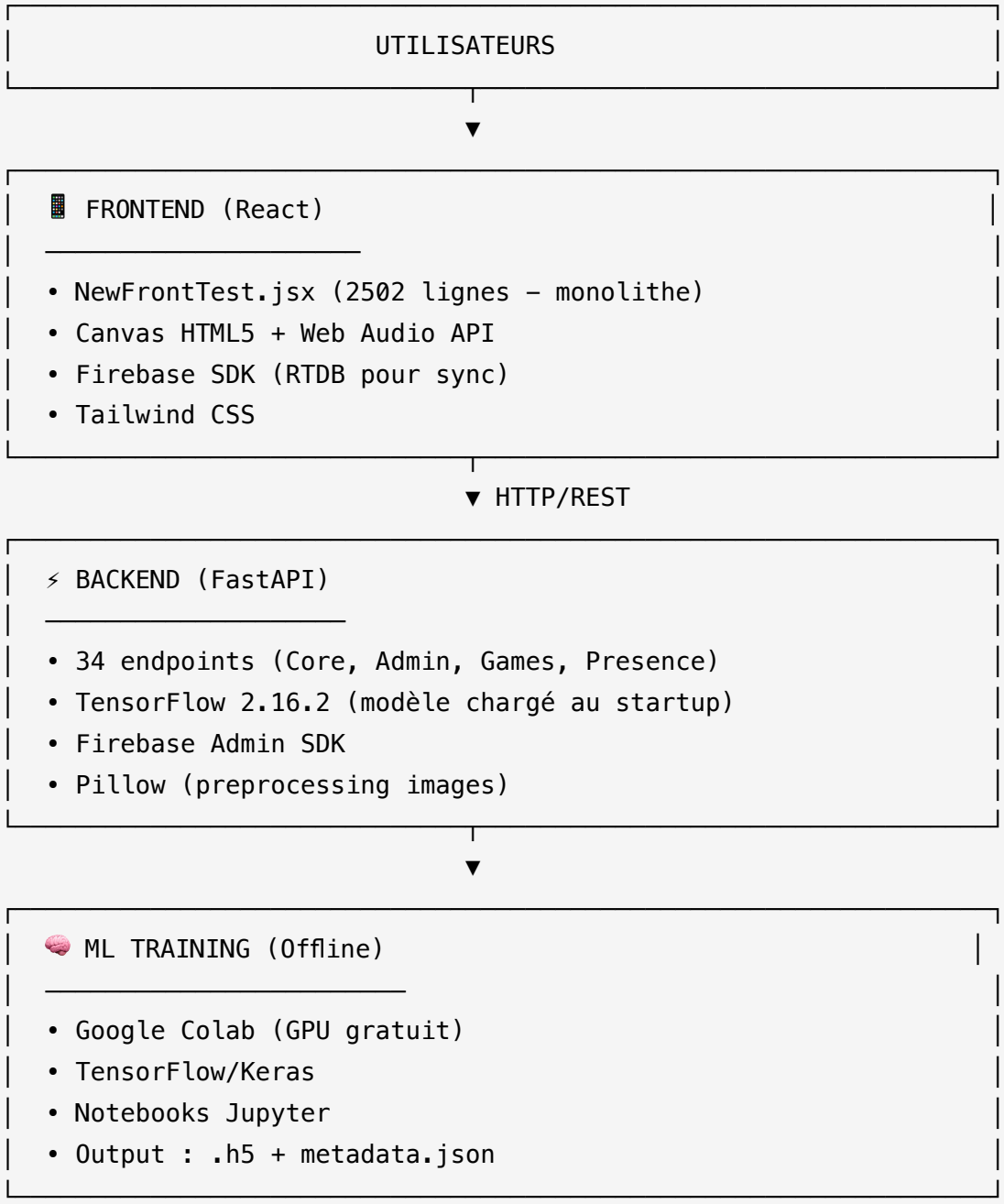
🎨 Visuel

- Diagramme 3 colonnes : Frontend | Backend | ML
- Flèches montrant les interactions

- Technologies par couche

Texte

ARCHITECTURE GLOBALE



SÉPARATION DES RESPONSABILITÉS

```
Frontend : UI, interactions, sync temps réel
Backend  : Inférence, API, logique métier
ML       : Entraînement offline, génération modèles
```

🎤 Script présentateur

> "Notre architecture se divise en trois parties distinctes. Le frontend React gère l'interface utilisateur, le canvas de dessin, l'audio et la synchronisation temps réel via Firebase. Le backend FastAPI expose 34 endpoints et effectue l'inférence TensorFlow - le modèle est chargé une seule fois au démarrage pour des prédictions rapides. L'entraînement ML est fait offline sur Google Colab avec GPU gratuit, produisant un fichier .h5 et ses métadonnées. Cette séparation claire des responsabilités permet de travailler sur chaque partie indépendamment."

📚 Informations de fond

- **Frontend :** src
- **Backend :** backend
- **ML Training :** ml-training

❓ Questions potentielles

1. **"Pourquoi ne pas faire l'inférence côté client (TensorFlow.js) ?"**
   - Possible et prévu en Phase 3, mais backend centralisé simplifie Active Learning et garantit consistance
2. **"Pourquoi Colab plutôt qu'un serveur dédié ?"**
   - GPU gratuit suffisant pour entraînement ponctuel, pas de coût infrastructure
3. **"Le monolithe frontend est-il un problème ?"**
   - Trade-off conscient pour MVP ; refactoring prévu en Phase 3

📄 SLIDE 11 : Architecture Frontend

🎨 Visuel

- Structure du fichier NewFrontTest.jsx en sections
- State machine visuelle
- Composants inline listés

📝 Texte

```
ARCHITECTURE FRONTEND : MONOLITHE INTENTIONNEL
═══════════════════════════════════════════════

📄 NewFrontTest.jsx (2502 lignes)
═══════════════════════════════════

COMPOSANTS INLINE
─────────────────
• WelcomeScreen       (lignes ~100-200)   → Accueil + health check
• GameModeSelection   (lignes ~200-350)   → Choix Classic/Race/Team
• TransitionOverlay   (lignes ~350-450)   → Animations rounds
• MultiplayerFlow     (lignes ~450-800)   → Lobby + waiting
• PlayingScreen       (lignes ~800-2000)  → Canvas + jeu
• GameOverScreen      (lignes ~2000-2200) → Résultats

STATE MACHINE
─────────────
WELCOME → MODE_SELECT → LOBBY_FLOW → PLAYING → GAME_OVER
                ↑                                  │
                └──────────────────────────────────┘

JUSTIFICATION MONOLITHE
═══════════════════════
✅ État partagé entre tous les écrans (pas de Context/Redux)
✅ Transitions fluides (pas de re-mount)
✅ Développement rapide MVP
✅ Debugging facilité (tout au même endroit)
⚠️ Trade-off : Moins modulaire, fichier volumineux

COMPOSANTS EXTRAITS (réutilisables)
═══════════════════════════════════
• AudioSettings.jsx (~150 lignes) → Modal paramètres audio
• ConnectionStatus.jsx (~50 lignes) → Indicateur connexion
• Toast.jsx (~80 lignes) → Notifications
```

🎤 Script présentateur

> "Notre frontend utilise une architecture monolithique **intentionnelle**. Le fichier NewFrontTest.jsx contient tous les écrans en composants inline. Pourquoi ce choix ? Dans un jeu avec de nombreuses transitions d'état - accueil, sélection mode, lobby, jeu, résultats - partager l'état entre composants séparés nécessiterait Context API ou Redux. Avec un monolithe, tous les useState sont au même niveau, les transitions sont fluides sans re-mount, et le debugging est simplifié. Seuls trois composants sont extraits car vraiment réutilisables : AudioSettings, ConnectionStatus et Toast. C'est un trade-off assumé pour le MVP."

📚 Informations de fond

- **Fichier principal :** NewFrontTest.jsx
- **Composants extraits :** components
- **Services :** services

❓ Questions potentielles

1. **"2500 lignes, n'est-ce pas une mauvaise pratique ?"**
   - Trade-off conscient : simplicité état vs modularité ; adapté au MVP
2. **"Comment maintenez-vous ce fichier ?"**
   - Sections bien délimitées par commentaires, IDE avec code folding
3. **"Refactoring prévu ?"**
   - Oui en Phase 3, probablement avec Zustand pour state management léger

---

📄 SLIDE 12 : Architecture Backend

🎨 Visuel

- Structure des routers FastAPI
- Diagramme du chargement modèle au startup
- Liste des 34 endpoints par groupe

Texte

ARCHITECTURE BACKEND : FASTAPI
═══════════════════════════════

STRUCTURE
─────────

```
backend/
├── main.py              → App FastAPI + endpoints core (5)
├── routers/
│   ├── admin.py         → Endpoints admin (6)
│   └── games.py         → Endpoints multiplayer (24)
├── services/
│   ├── firestore_service.py  → Accès Firestore
│   ├── presence_service.py   → Gestion présence RTDB
│   └── storage_service.py    → Firebase Storage
├── models/              → Fichiers .h5 et metadata
└── middleware/
    └── rate_limit.py    → Protection anti-spam
```

CHARGEMENT MODÈLE AU STARTUP
═══════════════════════════

```
@app.on_event("startup")
async def load_model():
    global model, categories
    model = tf.keras.models.load_model(MODEL_PATH)
    categories = load_categories_from_metadata()
```

Avantage : Inférence 12-18ms (vs 2000ms si chargé par requête)

34 ENDPOINTS ORGANISÉS
═══════════════════════

Groupe	Count	Exemples
Core	5	/predict, /health, /categories
Admin	6	/admin/retrain, /admin/cleanup
Race Mode	8	/games/race/create, /join
Guessing Mode	11	/games/guessing/*, /chat
Presence	5	/games/presence/heartbeat

```
JUSTIFICATION FASTAPI
─────────────────────
• Async natif → Non-blocking pendant inférence TensorFlow
• Documentation auto → /docs Swagger généré
• Validation Pydantic → Erreurs attrapées avant code métier
```

🎤 Script présentateur

> "Le backend est structuré avec FastAPI. Le fichier main.py contient l'app et les 5 endpoints core. Les routers séparent la logique : admin pour les opérations de maintenance, games pour le multiplayer. Les services encapsulent l'accès aux données Firebase. Point crucial : le modèle TensorFlow est chargé au startup de l'application, pas à chaque requête. Cela garantit une inférence en 12-18ms au lieu de 2 secondes. Les 34 endpoints sont documentés automatiquement par FastAPI - visitez /docs pour le Swagger interactif."

📚 Informations de fond

- **main.py :** main.py
- **Routers :** routers
- **Services :** services

❓ Questions potentielles

1. **"Pourquoi FastAPI plutôt que Flask ?"**
   - Async natif (ASGI vs WSGI), documentation auto, validation Pydantic
2. **"Comment gérez-vous les erreurs TensorFlow ?"**
   - Try/catch autour de l'inférence, fallback avec message d'erreur explicite
3. **"Le modèle en mémoire globale est-il thread-safe ?"**
   - Oui, TensorFlow gère le parallélisme interne ; une instance modèle suffit

---

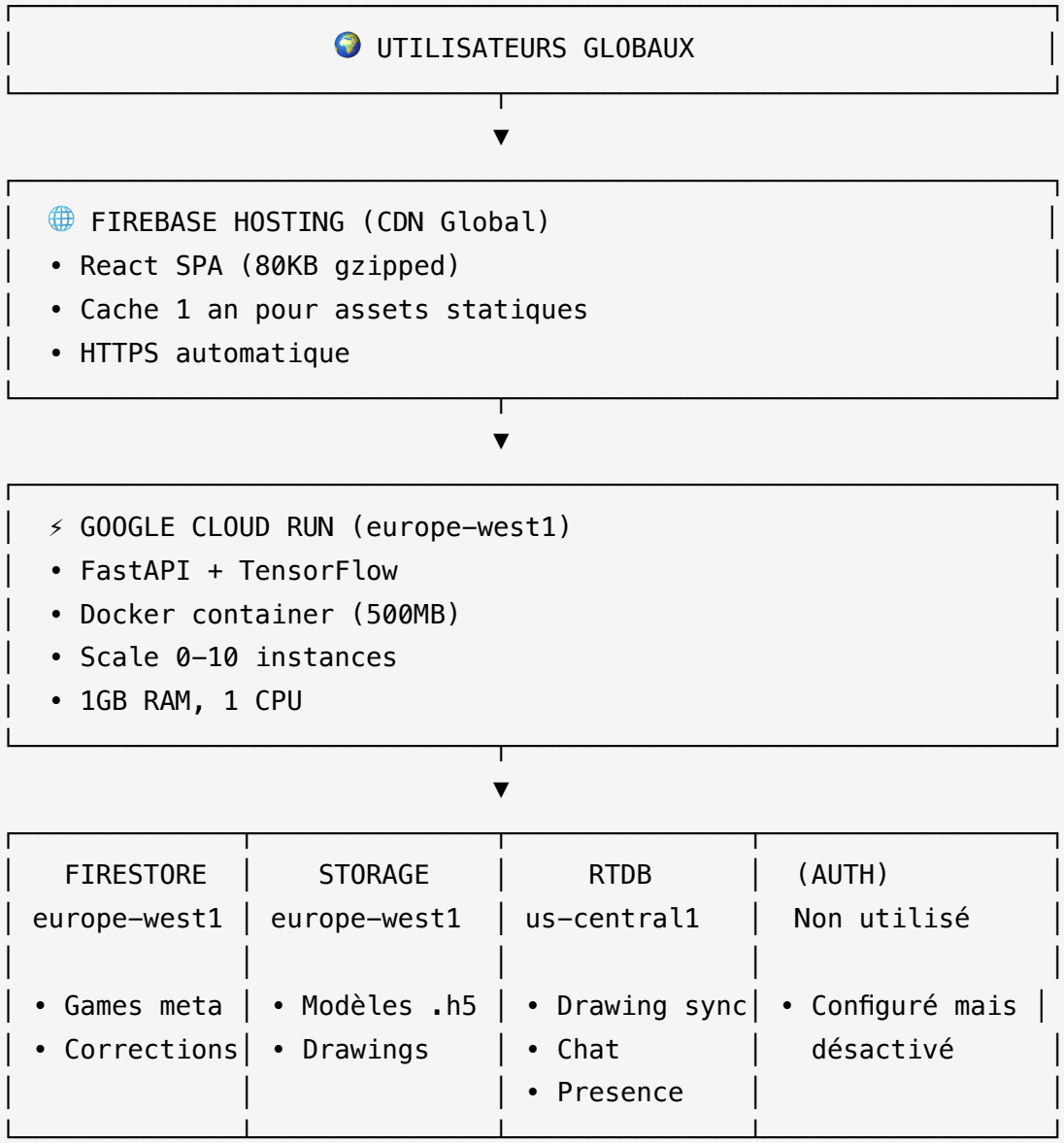
📄 SLIDE 13 : Architecture Cloud - Vue d'ensemble

🎨 Visuel

- Diagramme cloud avec tous les services
- Flèches montrant les communications
- Localisation géographique annotée

Texte

ARCHITECTURE CLOUD



LOCALISATION STRATÉGIQUE

- europe-west1 (Belgique) : Backend + Firestore + Storage
  → Latence ~30ms pour Europe
- us-central1 : RTDB (seule option free tier)
  → Acceptable car données éphémères
- CDN Global : Frontend distribué partout

🎤 Script présentateur

> "Notre architecture cloud utilise Firebase et Google Cloud. Le frontend est servi par Firebase Hosting avec CDN global - votre fichier JavaScript est distribué sur des serveurs partout dans le monde. Le backend tourne sur Cloud Run en europe-west1, choisi pour la latence européenne d'environ 30ms. Firestore et Storage sont co-localisés pour performance. La Realtime Database est en us-central1 - c'est la seule option du free tier, mais acceptable car les données sont éphémères. Point important : Firebase Auth est configuré mais **non utilisé** - les joueurs s'identifient par pseudo+emoji sans compte."

📚 Informations de fond

- **Configuration :** INFRASTRUCTURE.md
- **Déploiement :** deploy.sh
- **Variables :** env.yaml

❓ Questions potentielles

1. **"Pourquoi europe-west1 ?"**
   - Cible utilisateurs européens, latence ~30ms vs ~100ms depuis US
2. **"RTDB en us-central1 n'est pas un problème ?"**
   - Données éphémères (dessins temps réel), latence légèrement plus haute acceptable
3. **"Scale 0-10, que se passe-t-il au-delà ?"**
   - Quota Cloud Run, augmentable si besoin ; 10 instances suffisent pour milliers d'utilisateurs

---

📡 SLIDE 14 : Flux Réseau & Communications Inter-Services

🎨 Visuel

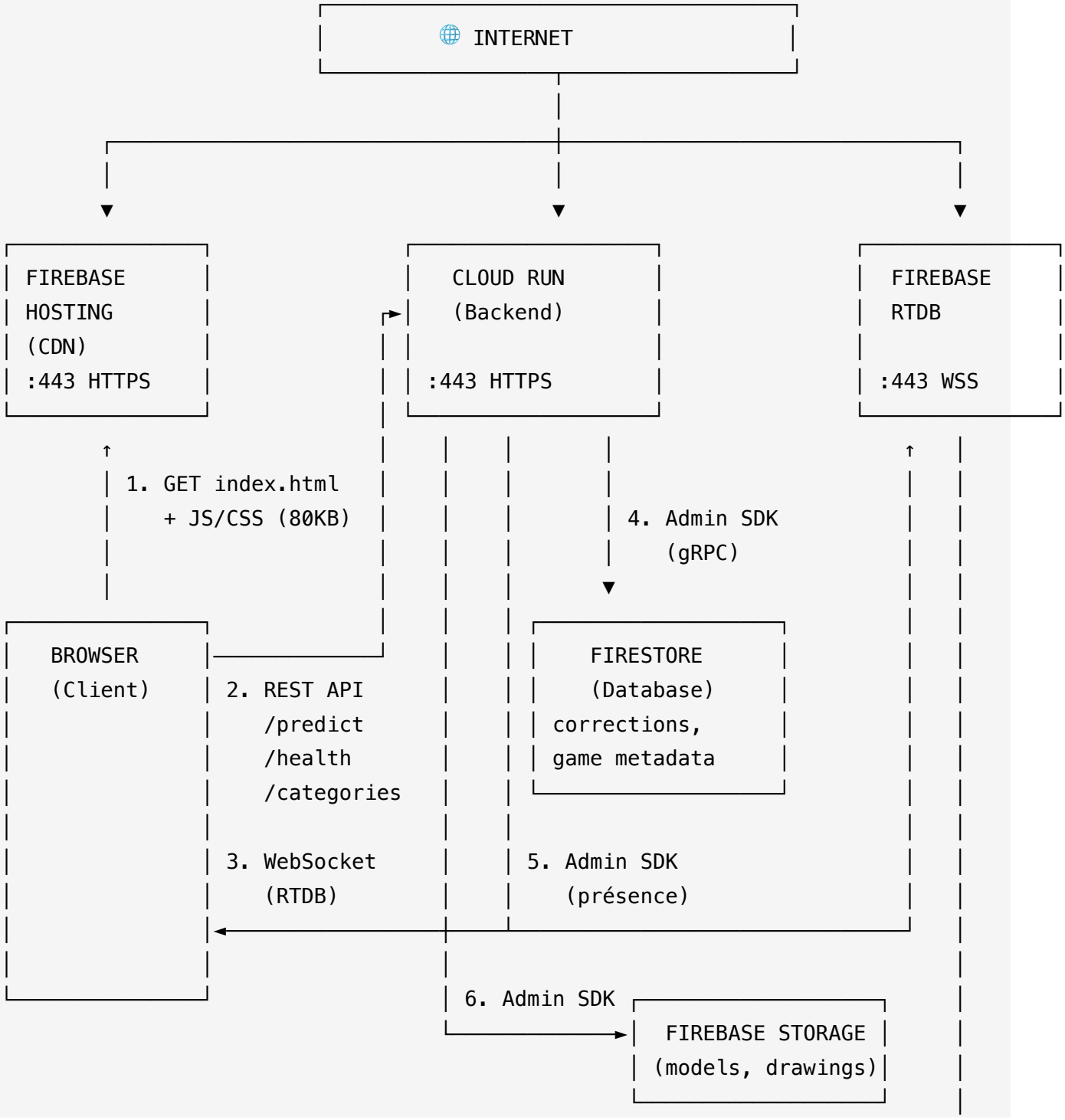
- Diagramme réseau complet avec tous les acteurs
- Flèches colorées par type de communication (HTTP, WebSocket, gRPC)

- Ports et protocoles annotés
- Tableau des requêtes par acteur

Texte

FLUX RÉSEAU : QUI PARLE À QUI ?

DIAGRAMME DES COMMUNICATIONS



DÉTAIL DES COMMUNICATIONS

Source	Destination	Protocole	But
Browser	Firebase Hosting	HTTPS GET	Charger SPA (index.html)
Browser	Cloud Run	HTTPS POST	/predict (inférence CNN)
Browser	Cloud Run	HTTPS GET	/health, /categories
Browser	Firebase RTDB	WebSocket (WSS)	Multiplayer: sync, chat, présence, état parties
Cloud Run	Firestore	gRPC (Admin SDK)	Sauvegarder corrections, metadata jeux
Cloud Run	Firebase RTDB	Admin SDK	Valider présence joueurs
Cloud Run	Firebase Storage	Admin SDK	Upload modèles/drawings (Active Learning)
(Scheduler)	Cloud Run	HTTPS POST	/admin/retrain (optionnel)

⚠️ NOTES:

- Le frontend N'UTILISE PAS Firestore ni Storage directement
- Tout le multiplayer passe par RTDB (temps réel)
- Cloud Scheduler est documenté mais non configuré en prod

🎤 Script présentateur

> "Détaillons les communications entre nos services. Le browser commence par charger l'application depuis Firebase Hosting via CDN - un GET HTTPS qui retourne notre SPA de 80KB. Pour les prédictions, le browser fait des requêtes POST vers Cloud Run sur l'endpoint /predict. **Point important** : tout le multiplayer temps réel utilise **exclusivement la Realtime Database** - création de parties, synchronisation du dessin, chat, présence des joueurs. Le frontend n'accède pas directement à Firestore ni Storage.

> Côté backend, Cloud Run communique avec Firestore via gRPC pour les corrections d'Active Learning, avec RTDB pour la gestion de présence, et avec Firebase Storage pour stocker les dessins et modèles. On a aussi préparé un endpoint /admin/retrain pour Cloud Scheduler, mais il n'est pas encore configuré en production."

📚 Informations de fond

- **Hosting :** CDN global, cache 1 an pour assets statiques
- **Cloud Run URL :** `https://ai-pictionary-backend-*.europe-west1.run.app`
- **RTDB WebSocket :** Connexion persistante via Firebase SDK (`onValue` , `set` , `update`)
- **Firestore :** Utilisé **uniquement par le backend** (corrections, metadata)
- **Storage :** Utilisé **uniquement par le backend** (modèles .h5, drawings PNG)
- **Frontend services :** `multiplayerService.js` (RTDB), `api.js` (REST Cloud Run)

❓ Questions potentielles

1. **"Pourquoi le frontend n'utilise pas Firestore ni Storage ?"**
   - RTDB suffit pour le multiplayer (latence 20-50ms), Storage/Firestore seraient redondants
   - Toutes les opérations persistantes passent par le backend (meilleur contrôle)
2. **"Le backend a-t-il besoin d'accéder à RTDB ?"**
   - Oui, pour valider la présence des joueurs et nettoyer les parties abandonnées (PresenceService)
3. **"Pourquoi gRPC entre Cloud Run et Firestore ?"**
   - Firebase Admin SDK utilise gRPC par défaut, plus performant que REST pour le serveur
4. **"Cloud Scheduler est-il configuré ?"**
   - Non en prod, mais l'endpoint `/admin/retrain` est prêt (documentation dans archive/)
5. *"Les endpoints /games/ dans Cloud Run sont-ils utilisés ?"**
   - Partiellement préparés mais le frontend utilise RTDB directement pour plus de réactivité

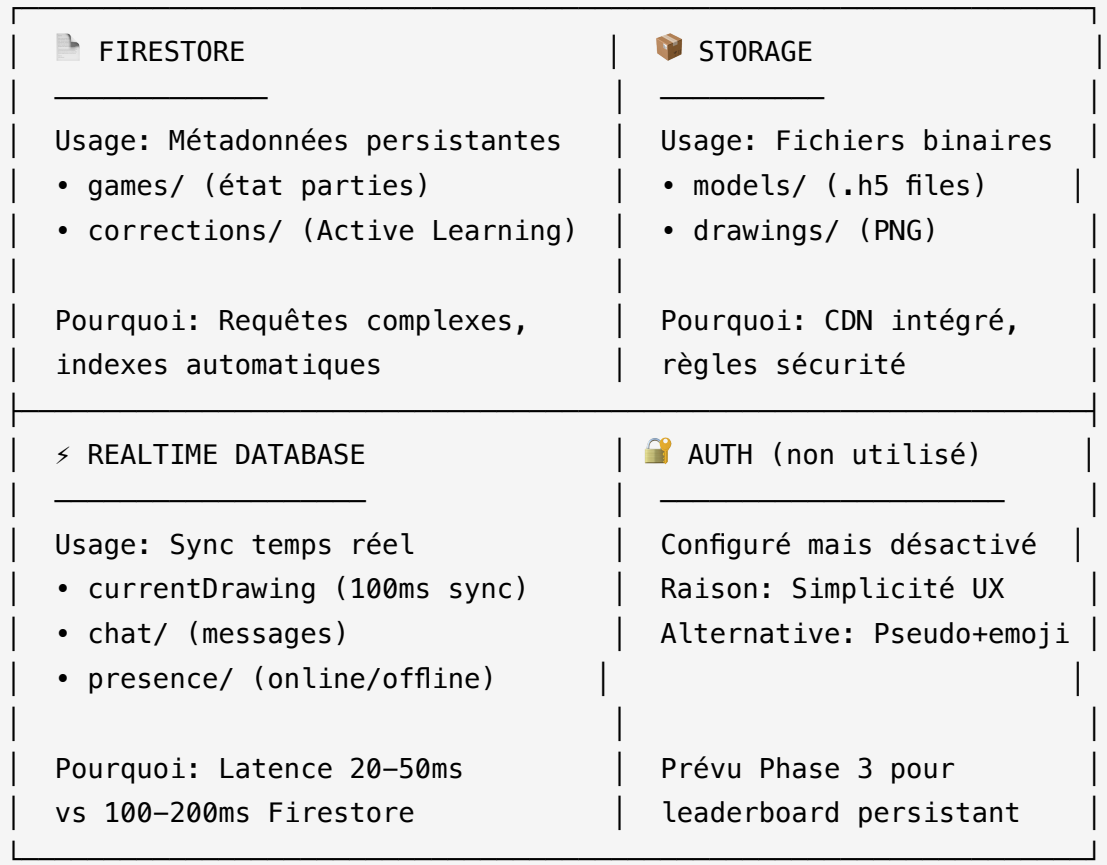
📄 SLIDE 15 : Firebase - Justification des Services

🎨 Visuel

- 4 cards pour chaque service Firebase utilisé
- Tableau comparatif Firebase vs alternatives
- Indicateur coût

📝 Texte

FIREBASE : POURQUOI ET COMMENT



COMPARAISON FIREBASE vs AWS

Aspect	Firebase ✅	AWS
Setup	5 minutes	30+ minutes
Real-time sync	Built-in	WebSocket DIY
Coût (100 DAU)	~$0	~$5/mois
SDK Frontend	Intégré	Amplify (lourd)
Documentation	Excellente	Complexe

VERDICT: Firebase = Développement 5x plus rapide pour ce cas

🎤 Script présentateur

> "Pourquoi Firebase ? Firestore stocke nos métadonnées de parties et les corrections pour l'Active Learning - ses requêtes complexes et indexes automatiques sont idéaux. Storage héberge les modèles et dessins avec un CDN intégré. La **Realtime Database** est notre choix clé pour le multijoueur : sa latence de 20-50ms permet de synchroniser le canvas du dessinateur vers les spectateurs en temps réel. Firestore serait trop lent à 100-200ms. Comparé à AWS, Firebase nous a fait gagner un temps considérable : le SDK frontend est intégré, le real-time est built-in, et le coût est quasi nul. C'est du développement 5 fois plus rapide pour notre cas d'usage."

📚 Informations de fond

- **RTDB structure :** INFRASTRUCTURE.md
- **Firestore collections :** games/, corrections/
- **Comparatif détaillé :** TECHNICAL_REFERENCE.md

❓ Questions potentielles

1. **"Pourquoi ne pas tout mettre dans RTDB ?"**
   - RTDB = arbre JSON, pas de requêtes complexes ; Firestore meilleur pour données structurées
2. **"Le vendor lock-in Firebase n'est-il pas risqué ?"**
   - Acceptable pour projet académique ; migration possible vers Supabase si besoin
3. **"Auth non utilisé, pourquoi l'avoir configuré ?"**
   - Préparé pour extension future (leaderboard), configuration rapide

---

📄 SLIDE 16 : Google Cloud Run - Justification

🎨 Visuel

- Tableau comparatif Cloud Run vs Functions vs App Engine
- Schéma scale-to-zero
- Configuration déploiement

Texte

GOOGLE CLOUD RUN : POURQUOI CE CHOIX

COMPARAISON OPTIONS GCP

Critère	Cloud Run ✅	Cloud Functions	App Engine
Container	✅ Docker	❌ Buildpacks	⚠️ Config
Mémoire max	32 GB	16 GB	10 GB
TensorFlow 500MB	✅ Facile	⚠️ Complexe	⚠️ Complexe
Cold start	5-8s prévis	3-8s variable	N/A
Scale-to-zero	✅ $0 idle	✅ $0 idle	❌ $25+/mois
Contrôle startup	✅ on_event	❌ Per-request	✅

POURQUOI CLOUD RUN ?

1. DOCKER = Environnement identique local/prod
   - TensorFlow 2.16.2 + modèle = 500MB image
   - Fonctionne sur mon laptop = Fonctionne en prod

2. STARTUP EVENT = Modèle chargé UNE FOIS
   - @app.on_event("startup") → load_model()
   - Inférence 12-18ms vs 2000ms si lazy load

3. SCALE-TO-ZERO = Coût ~$0
   - min-instances: 0 → Pas de coût au repos
   - max-instances: 10 → Absorbe les pics

4. COLD START PRÉVISIBLE
   - 5-8 secondes constant
   - vs Cloud Functions 3-8s aléatoire

CONFIGURATION PRODUCTION

```
--region europe-west1
--memory 1Gi
--cpu 1
```

```
--min-instances 0
--max-instances 10
--timeout 60s
--allow-unauthenticated
```

🎤 Script présentateur

> "Cloud Run s'est imposé face aux alternatives. Cloud Functions ne supporte pas bien les containers custom - notre image Docker de 500MB avec TensorFlow aurait été problématique. App Engine ne scale pas à zéro, donc coût fixe de $25/mois même sans trafic. Cloud Run offre le meilleur des deux mondes : container Docker pour environnement identique local/prod, startup event pour charger le modèle une seule fois, et scale-to-zero pour un coût quasi nul. Le cold start de 5-8 secondes est prévisible, contrairement à Cloud Functions qui varie. Notre configuration utilise 1GB de RAM, suffisant pour TensorFlow, avec scaling automatique jusqu'à 10 instances."

📚 Informations de fond

- **Dockerfile :** backend/Dockerfile
- **Configuration :** backend/env.yaml
- **Commande deploy :** INFRASTRUCTURE.md

❓ Questions potentielles

1. **"5-8s de cold start n'est pas trop long ?"**
   - Acceptable pour projet académique ; option min-instances=1 pour $5/mois si critique
2. **"Pourquoi pas Kubernetes (GKE) ?"**
   - Overkill pour notre échelle, coût de base élevé, complexité non justifiée
3. **"L'image 500MB n'est-elle pas trop grosse ?"**
   - TensorFlow incompressible ; possible d'utiliser TF Lite pour réduire mais perte de features

---

📄 SLIDE 17 : Flux de Données - Mode Classic

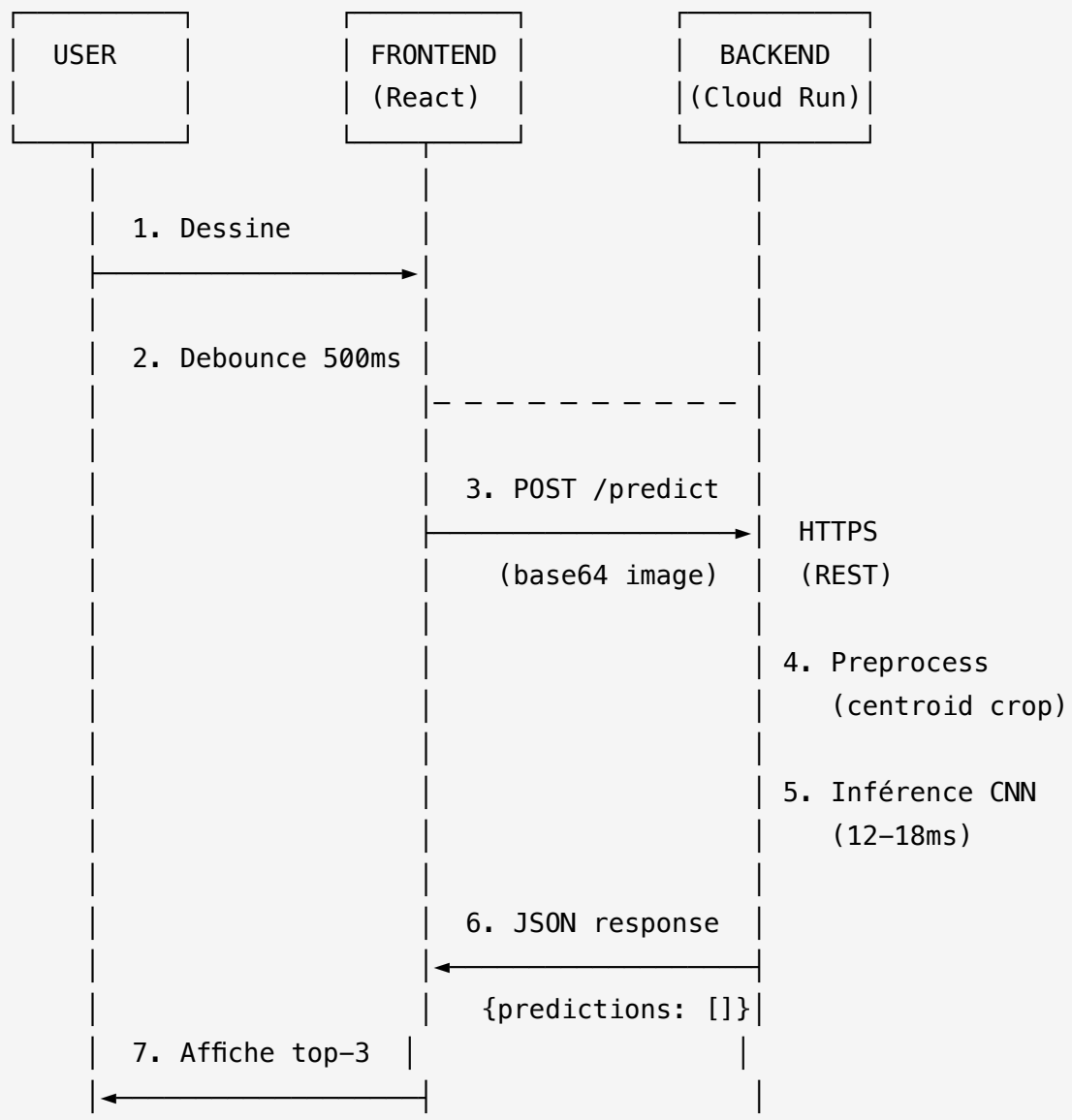
🎨 Visuel

- Diagramme de séquence simplifié
- Flèches avec latences annotées
- États du frontend colorés

Texte

FLUX DE DONNÉES : MODE CLASSIC (SOLO)

SÉQUENCE COMPLÈTE (voir slide 14 pour architecture réseau)



REQUÊTE /predict

```
POST https://ai-pictionary-backend-*.run.app/predict
Content-Type: application/json

{
```

```
  "image": "..."
}

RÉPONSE
────────
{
  "predictions": [
    {"category": "cat", "confidence": 0.92},
    {"category": "dog", "confidence": 0.05},
    {"category": "bear", "confidence": 0.02}
  ],
  "inference_time_ms": 15
}

LATENCES MESURÉES
──────────────────
• Debounce      : 500ms (côté client)
• Network RTT   : 50-100ms (europe-west1)
• Preprocess    : 5-10ms
• Inférence     : 12-18ms
• Total E2E     : 120-350ms
```

🎤 Script présentateur

> "Voyons le flux de données en mode Classic. L'utilisateur dessine sur le canvas. Après un debounce de 500ms - c'est-à-dire 500ms sans nouveau trait - le frontend envoie l'image en base64 au backend. Le backend préprocesse avec centroid crop, effectue l'inférence CNN en 12-18ms, et retourne les prédictions en JSON. Le frontend affiche le top-3. Le temps total end-to-end est de 120 à 350ms. Pourquoi 500ms de debounce ? Trop court et on spam l'API inutilement. Trop long et le feedback est frustrant. 500ms correspond à la pause naturelle entre deux traits quand on dessine."

📚 Informations de fond

- **API endpoint :** `POST /predict` dans main.py
- **Frontend debounce :** Implémenté avec setTimeout dans NewFrontTest.jsx
- **Format réponse :** `{predictions: [{category, confidence}], ...}`

❓ Questions potentielles

1. **"Pourquoi ne pas utiliser WebSocket pour les prédictions ?"**
   - REST suffisant avec debounce ; WebSocket ajoute complexité sans gain significatif
2. **"Le debounce est-il configurable ?"**
   - Actuellement hardcodé ; pourrait être un setting utilisateur
3. **"Que se passe-t-il si le backend ne répond pas ?"**
   - Timeout côté frontend, affichage message d'erreur, retry automatique au prochain trait

---

📄 SLIDE 18 : Flux de Données - Mode Multiplayer

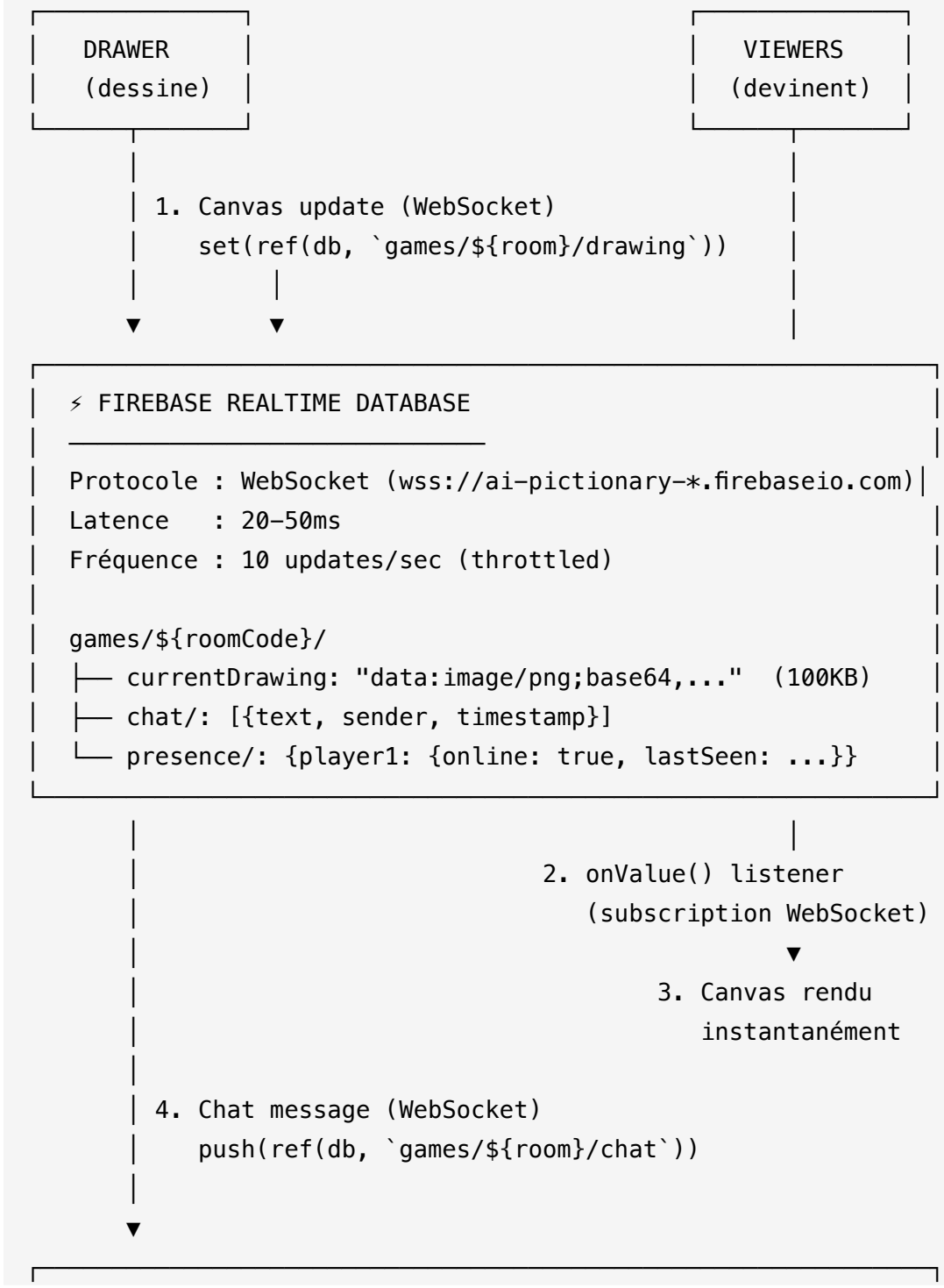
🎨 Visuel

- Diagramme avec multiple joueurs
- Distinction Firestore vs RTDB avec couleurs
- Sync temps réel visualisé

Texte

FLUX DE DONNÉES : MODE TEAM VS IA

COMMUNICATIONS RÉSEAU DÉTAILLÉES



```
│  🎯 CLOUD RUN (pour prédiction IA)                                  │
│  ──────────────────────────────────                                 │
│  Protocole : HTTPS REST                                             │
│                                                                     │
│  5. POST /games/guessing/ai-prediction                              │
│     Body: {game_id, image}                                          │
│     → CNN infère, compare avec catégorie cible                      │
│     → Si match 85%+ → IA gagne le round                             │
└─────────────────────────────────────────────────────────────────────┘
        │
        │ 6. Update game state (gRPC)
        ▼
┌─────────────────────────────────────────────────────────────────────┐
│  📄 FIRESTORE                                                       │
│  ─────────────                                                      │
│  Protocole : SDK (gRPC interne)                                     │
│  Latence   : 100-200ms                                              │
│                                                                     │
│  games/{gameId}/                                                    │
│  ├── scores: {player1: 5, player2: 3, ai: 2}                        │
│  ├── currentRound: 3                                                │
│  └── status: "playing"                                              │
└─────────────────────────────────────────────────────────────────────┘

RÉSUMÉ DES PROTOCOLES UTILISÉS
──────────────────────────────

┌──────────────────────┬──────────────┬──────────────────────────────┐
│ Communication        │ Protocole    │ Pourquoi                     │
├──────────────────────┼──────────────┼──────────────────────────────┤
│ Drawer → RTDB        │ WebSocket    │ Haute fréquence (10/sec)     │
│ RTDB → Viewers       │ WebSocket    │ Push temps réel              │
│ Client → Cloud Run   │ HTTPS REST   │ Requêtes ponctuelles         │
│ Cloud Run → RTDB     │ REST Admin   │ Validation serveur           │
│ Cloud Run → Firestore│ gRPC         │ Performance backend          │
└──────────────────────┴──────────────┴──────────────────────────────┘
```

🎤 Script présentateur

> "Le mode Team vs IA illustre la complexité de nos communications. Le dessinateur

> envoie son canvas via WebSocket vers RTDB toutes les 100ms - c'est du push bidirectionnel maintenu par le SDK Firebase. Les viewers reçoivent ces updates quasi-instantanément via leur propre listener WebSocket. Pour la prédiction IA, une requête HTTPS classique part vers Cloud Run qui effectue l'inférence et détermine si l'IA a trouvé. Cloud Run met ensuite à jour les scores dans Firestore via gRPC, le protocole binaire du SDK Admin. Chaque protocole est choisi pour son cas d'usage : WebSocket pour le temps réel haute fréquence, REST pour les opérations CRUD, gRPC pour les communications backend-to-backend performantes."

📚 Informations de fond

- **Service RTDB :** multiplayerService.js
- **Structure RTDB :** INFRASTRUCTURE.md
- **Présence :** presence_service.py

❓ Questions potentielles

1. **"100ms d'update, n'est-ce pas trop fréquent ?"**
   - Compromis fluidité vs bande passante ; compression PNG aide
2. **"Comment gérez-vous les conflits de données ?"**
   - RTDB = last-write-wins pour dessin (un seul drawer), Firestore transactions pour scores
3. **"Que se passe-t-il si un joueur perd la connexion ?"**
   - Système de présence détecte via heartbeat, marque offline après 30s

---

📄 SLIDE 19 : State Machine du Jeu

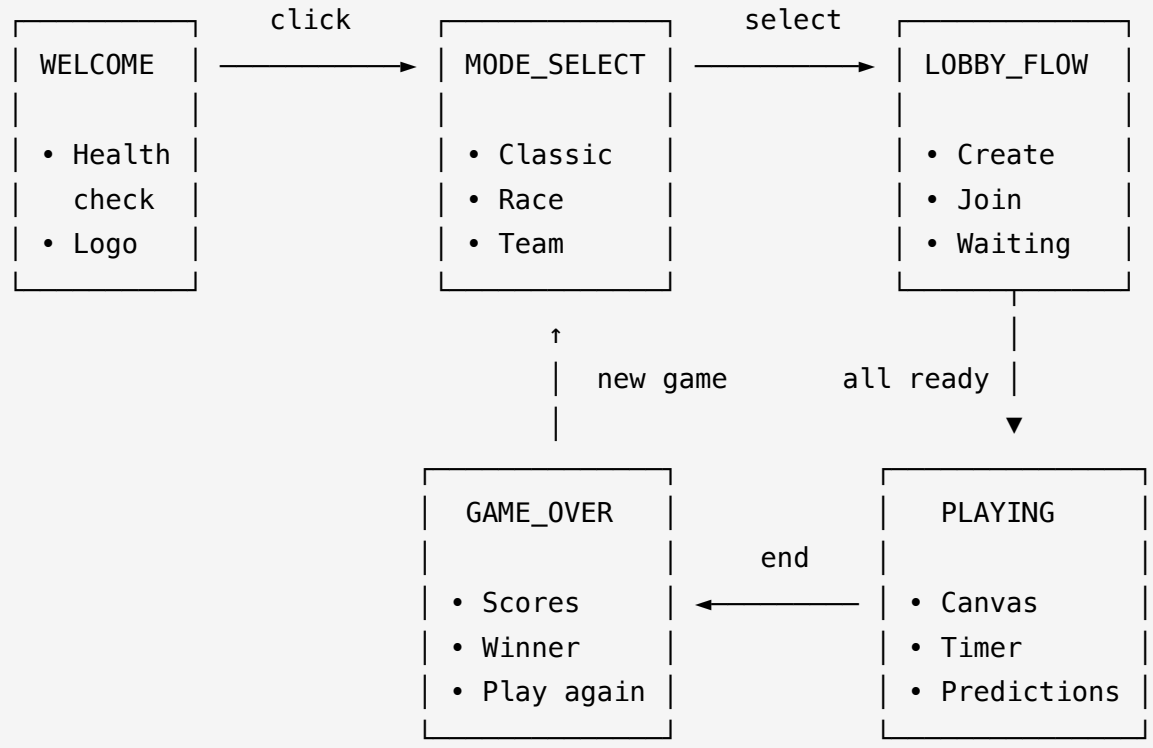
🎨 Visuel

- Diagramme d'états avec transitions
- Conditions de transition annotées
- Exemple de flow complet

Texte

STATE MACHINE : FLOW DU JEU
═══════════════════════════

ÉTATS PRINCIPAUX (gameState)
────────────────────────────



MODES DE JEU (gameMode)
───────────────────────

• CLASSIC → Boucle PLAYING seulement (pas de LOBBY)
• RACE   → Compétition, tous dessinent simultanément
• TEAM   → Coopératif, 1 drawer + guessers vs IA

TRANSITIONS CRITIQUES
─────────────────────

LOBBY_FLOW → PLAYING : Quand tous les joueurs sont "ready"
PLAYING → PLAYING   : Nouveau round (même partie)
PLAYING → GAME_OVER : Dernier round terminé ou timeout
GAME_OVER → WELCOME : Bouton "New Game"

🎤 Script présentateur

> "Voici la state machine qui orchestre le jeu. Cinq états principaux : WELCOME pour l'accueil avec health check backend, MODE_SELECT pour choisir entre Classic, Race ou Team, LOBBY_FLOW pour la création/jonction de partie et l'attente des joueurs, PLAYING pour le jeu actif, et GAME_OVER pour les résultats. Les transitions sont déclenchées par des actions utilisateur ou des événements : tous les joueurs ready déclenche le passage au jeu, la fin du dernier round déclenche Game Over. En mode Classic, on saute directement de MODE_SELECT à PLAYING sans passer par le lobby."

📚 Informations de fond

- **Implémentation :** Variable `gameState` dans NewFrontTest.jsx
- **Conditions :** Vérifications dans useEffect et handlers d'événements
- **Persistance :** État en mémoire uniquement (refresh = retour WELCOME)

❓ Questions potentielles

1. **"L'état est-il persisté en cas de refresh ?"**
   - Non, retour à WELCOME ; acceptable pour party game, localStorage possible si besoin
2. **"Comment gérez-vous les déconnexions en mode PLAYING ?"**
   - Détection via presence, partie peut continuer ou mettre en pause selon nombre de joueurs restants
3. **"Pourquoi pas une librairie de state machine (XState) ?"**
   - Overhead pour 5 états ; useState + conditions suffisent pour cette complexité

📄 SLIDE 20 : Démonstration Live & Conclusion

🎨 Visuel

- QR Code grand format
- Récapitulatif visuel du parcours (ML → Cloud → App)
- Screenshots des 3 modes

Texte

DÉMONSTRATION LIVE

🔗 ai-pictionary-4f8f2.web.app

[QR CODE]

RÉCAPITULATIF DU PARCOURS

📊	DATASET	→ Quick Draw reconstitué, centroid crop (+3%)
🧠	CNN	→ 3 Conv + Dense 128, 90.2% accuracy, 12-18ms
☁️	CLOUD	→ Firebase + Cloud Run, ~$0/mois
🌐	RÉSEAU	→ REST + WebSocket + gRPC selon le besoin
🎮	APPLICATION	→ 3 modes, 34 endpoints, state machine claire

TOUS LES CHOIX JUSTIFIÉS

✓	Raster vs Vectoriel	→ CNN simple, latence prévisible
✓	Centroid crop	→ +3.1% accuracy, coût nul
✓	50 classes (v4)	→ Équilibre variété/précision
✓	FastAPI	→ Async, docs auto, Pydantic
✓	Cloud Run	→ Docker, scale-to-zero, startup event
✓	RTDB + Firestore	→ Temps réel + Persistance
✓	WebSocket (RTDB)	→ 10 updates/sec pour dessin
✓	REST (Cloud Run)	→ CRUD parties et prédictions
✓	gRPC (Firestore)	→ Backend-to-backend performant
✓	Monolithe frontend	→ État partagé, dev rapide

CE QUE NOUS AVONS DÉMONTRÉ

1. Pipeline ML complet : Data → Training → Inference
2. Architecture cloud moderne et scalable
3. Communications optimisées par cas d'usage
4. Application interactive temps réel
5. Choix techniques défendables et justifiés

```
MERCI ! QUESTIONS ?
```

🎤 Script présentateur

> "Je vous invite à tester l'application en scannant ce QR code. Récapitulons notre parcours : nous avons reconstitué le dataset Quick Draw avec notre optimisation centroid crop, entraîné un CNN à 90.2% d'accuracy avec inférence en 12-18ms, déployé sur une architecture cloud Firebase + Cloud Run pour environ 0 dollar par mois, et développé une application avec 3 modes de jeu et audio synthétique. Chaque choix technique a été justifié : raster pour simplifier le CNN, centroid crop pour normaliser les dessins, Cloud Run pour Docker et scale-to-zero, dual-database pour temps réel et persistance. Nous avons démontré un pipeline ML complet, de la donnée brute à l'application interactive. Merci de votre attention, je suis prêt pour vos questions !"

📚 Informations de fond

- **URL production :** https://ai-pictionary-4f8f2.web.app
- **Toute la documentation :** docs
- **Code source :** Repository GitHub

❓ Questions potentielles

1. **"Quel a été le plus grand défi ?"**
   - Synchronisation temps réel Team vs IA avec latence <50ms ; résolu avec RTDB
2. **"Que feriez-vous différemment ?"**
   - Architecture frontend modulaire dès le départ, TensorFlow.js pour inférence client
3. **"Prochaines étapes ?"**
   - Active Learning UI, refactoring frontend, leaderboard persistant, version mobile

---

📋 RÉCAPITULATIF DE LA STRUCTURE

#	Slide	Focus
1	Titre	Introduction

#	Slide	Focus
2	Contexte Quick Draw	Inspiration
3	Dataset Quick Draw	Données source
4	Reconstruction Dataset	Notre preprocessing
5	Fonctionnement CNN	Théorie
6	Architecture CNN	Notre modèle
7	Validation Hyperparamètres	Expérimentations
8	Résultats v4.0.0	Métriques
9	Comparaison Versions	Trade-offs
10	Architecture Globale	Vue d'ensemble
11	Architecture Frontend	Monolithe
12	Architecture Backend	FastAPI
13	Architecture Cloud	Vue d'ensemble services
14	Flux Réseau	Communications inter-services
15	Firebase Services	Justification (14)
16	Cloud Run	Justification (15)
17	Flux Classic	Données mode solo (16)
18	Flux Multiplayer	Dual-database (17)
19	State Machine	États du jeu (18)
20	Démo & Conclusion	Récapitulatif (20)