ChatGPT

Plan d'implémentation du réarrangement spatial sémantique des nœuds

Pipeline backend : API Next.js et analyse sémantique des relations

Pour réaliser le réarrangement automatique, nous allons créer une route API dédiée côté backend (Next.js). Cette route recevra la liste des nœuds d'un groupe (au minimum l'`id` et le contenu textuel de chaque nœud) et retournera un nouvel agencement (positions, et éventuellement liens entre nœuds). Le fonctionnement du pipeline sera le suivant :

- **(a) Endpoint API** : Créez une route API POST (par exemple `src/app/api/arrange/route.ts`) qui gère la requête de réorganisation. À l'appel, l'API récupère depuis `req.json()` la liste des nœuds envoyés (ex. `{ nodes: [ {id: 'node-1', content: '...'}, ...] }`). Il faut sécuriser cet endpoint comme les autres endpoints AI du projet : vérifier l'utilisateur via le cookie `uid` et ses droits (plan/tokens) avant de procéder [1] [2]. Par exemple :

```
// src/app/api/arrange/route.ts
import { NextResponse } from 'next/server';
import { cookies } from 'next/headers';
import prisma from '@/lib/prisma';
import { getUserSubscriptionPlan } from '@/utils/subscription';

export async function POST(req: Request) {
  const cookieStore = cookies();
  const uid = cookieStore.get('uid')?.value;
  if (!uid) return NextResponse.json({ error: 'Unauthorized' }, { status:
401 });
  const user = await prisma.user.findUnique({ where: { id: uid } });
  if (!user) return NextResponse.json({ error: 'User not found' }, { status:
404 });
  const plan = getUserSubscriptionPlan(user);
  if (plan.aiTokens === 0 || user.aiTokensUsed >= plan.aiTokens) {
    return NextResponse.json({ error: 'Plan AI épuisé' }, { status: 403 });
  }

  const { nodes } = await req.json();
  // ... (logique de réarrangement détaillée ci-dessous)
}
```

- **(b) Analyse sémantique des relations** : L'API utilise un petit modèle de *ranking/classification* (non-LLM) pour identifier les relations entre chaque paire de nœuds de texte. L'objectif est de déterminer : **quel nœud est central** (ex. le point de départ ou la question initiale) et **quels**

nœuds sont des réponses ou des oppositions par rapport à ce nœud central. Pour ce faire, on peut procéder ainsi :

- **Vectorisation & similarité** – Convertir le texte de chaque nœud en vecteur d'embedding sémantique. Par exemple, utiliser un modèle *Sentence Transformer* léger (6 couches) comme `all-MiniLM-L6-v2` pour obtenir un vecteur par texte. En comparant ces vecteurs, on peut estimer la proximité sémantique entre nœuds. Le nœud ayant le contenu le plus « central » sera souvent celui avec la plus grande similarité moyenne vis-à-vis des autres [3]. On désigne ce nœud comme **nœud principal** (éventuellement le « début » de la discussion).
- **Classification des relations** – Pour chaque autre nœud, on détermine s'il s'agit d'une **réponse** (contenu cohérent ou en accord avec le nœud principal) ou d'une **opposition** (contenu contradictoire ou en désaccord). Un petit modèle de classification binaire ou NLI (Natural Language Inference) peut faire cela. Par exemple, un modèle DistilBERT fine-tuné sur MNLI (entailment/contradiction) peut servir à étiqueter la relation entre le texte principal (premise) et le texte du nœud secondaire (hypothesis) [4]. S'il y a contradiction détectée, on tague le nœud comme **opposition**. Sinon (entailment ou neutre), on le considère comme **réponse/support** du nœud principal. Ce modèle est léger (~67M paramètres) et disponible sur HuggingFace (`typeform/distilbert-base-uncased-mnli`), ce qui permet un déploiement peu coûteux comparé à un LLM. On peut le charger dans notre API route via Transformers :

```
// Chargement du tokenizer et modèle (une seule fois en haut du fichier
route.ts par ex.)
import { AutoTokenizer, AutoModelForSequenceClassification } from
'@huggingface/transformers';
const tokenizer = AutoTokenizer.from_pretrained("typeform/distilbert-base-
uncased-mnli");
const nliModel =
AutoModelForSequenceClassification.from_pretrained("typeform/distilbert-base-
uncased-mnli");
...
// Dans la logique de classification:
const inputs = tokenizer(`${mainText} </s> ${otherText}`, { returnTensors:
'tf' /* ou 'pt' */});
const result = await nliModel.predict(inputs);
// Analyser result pour voir si "contradiction" vs "entailment/neutral"
```

(Dans le cas où on ne peut pas charger le modèle directement sur Vercel, on pourra appeler l'API HuggingFace Inference pour obtenir le label de relation, ou utiliser une version distillée en ONNX/TF.js pour exécution côté serveur ou Edge.)

- **(c) Génération du layout structuré** : À partir du nœud principal identifié et des catégories de chaque nœud secondaire (réponse ou opposition), le backend construit une structure hiérarchique en arbre. Concrètement, on va calculer de nouvelles coordonnées pour chaque nœud afin de les disposer visuellement de façon organisée :
- Placer le **nœud principal** en haut (par exemple, vers le haut-centre de la zone du groupe). On peut choisir une marge (ex. `{x: 100, y: 50}` relativement à l'origine du groupe) pour le point de départ.
- Placer les **nœuds "réponse"** directement en dessous du principal, en colonne verticale. Chaque réponse peut être décalée légèrement en *x* pour bien les centrer sous le principal. Par exemple, si le nœud principal fait 220px de large, on peut aligner les réponses à la même abscisse, avec

un décalage en *y* de ~150px entre chaque réponse pour éviter le chevauchement. S'il y a plusieurs réponses, on obtient une pile verticale.

- Placer les **nœuds "opposition"** sur les côtés du nœud principal (droite et/ou gauche). Par exemple, avec une opposition, on la positionnera à droite du nœud principal, à peu près au même niveau vertical (pour signifier qu'elle s'oppose directement à l'énoncé principal). S'il y a plusieurs oppositions, on peut en placer une à gauche et une à droite pour équilibrer, ou les étager latéralement. L'idée est de représenter visuellement les objections sur les côtés : dans un schéma argumentatif, le point central est en haut, les arguments de soutien en dessous (flèches vertes), et les objections en rouge sur le côté. Notre agencement suivra ce principe en répartissant les nœuds opposants à gauche/droite du nœud central et légèrement décalés en *y* si nécessaire (pour qu'ils ne se chevauchent pas entre eux ni avec le principal).
- (Optionnel) **Liens/Arêtes** : On peut aussi générer des arêtes (*edges*) entre les nœuds pour représenter explicitement les relations trouvées. Par exemple, on créera un edge du nœud principal vers chaque réponse, et un edge du nœud principal vers chaque opposition (éventuellement avec un style différent ou un label pour indiquer l'opposition). Ces edges permettront de visualiser la structure en arbre dans le whiteboard (React Flow supporte les arêtes connectant les nodes). On pourra donner un `source` (id du principal) et un `target` (id du secondaire) pour chaque lien, puis renvoyer la liste des edges également.
- L'API compile ainsi le résultat sous forme d'un objet JSON, par ex. `{ positions: [ {id: 'node-1', x: 100, y: 50}, {id: 'node-2', x: 120, y: 220}, ... ], edges: [ { source: 'node-1', target: 'node-2'}, ...] }`.

Ce calcul se fera dans la suite de notre fonction route API. En résumé, après la phase (b) on a : `mainId` (id du nœud principal), un tableau des `responsesIds` et `oppositionsIds`. On peut alors appliquer une logique de placement simple, par exemple :

```
// suite dans src/app/api/arrange/route.ts
const relations = classifyRelations(nodes); // fonction qui retourne mainId, responses, oppositions
const mainId = relations.mainId;
const responses = relations.responses;      // array of node IDs
const oppositions = relations.oppositions;  // array of node IDs

// Dimensions approximatives
const mainWidth = 220, nodeHeight = 100;
const marginX = 40, marginY = 40;
let layout = [];

// Place main node
layout.push({ id: mainId, x: 200, y: 50 });

// Place response nodes in a vertical column below main
responses.forEach((nodeId, idx) => {
  layout.push({
    id: nodeId,
    x: 200,
    y: 50 + (idx+1) * (nodeHeight + marginY)  // stack vertically
  });
  edges.push({ source: mainId, target: nodeId });
});
```

```
// Place opposition nodes to the sides of main
oppositions.forEach((nodeId, idx) => {
  const side = (idx % 2 === 0) ? 'right' : 'left';
  const offsetX = side === 'right' ? mainWidth + marginX : -(mainWidth +
marginX);
  const offsetY = (Math.floor(idx/2) + 0.5) * (nodeHeight + marginY);
  layout.push({
    id: nodeId,
    x: 200 + offsetX,
    y: 50 + offsetY
  });
  edges.push({ source: mainId, target: nodeId, type: 'opposition' });
});
return NextResponse.json({ positions: layout, edges });
```

(Le code ci-dessus illustre la logique de placement : on centre les réponses sous le nœud principal, et on distribue les oppositions à gauche et à droite. On a ajouté un champ `type: 'opposition'` *pour les edges d'opposition, ce qui permettra côté frontend de styliser différemment si on le souhaite.)*

L'API renvoie donc ces nouvelles coordonnées et liens. Le backend n'a pas besoin de connaître la position absolue du groupe sur le canvas : les positions calculées peuvent être traitées comme **coordonnées relatives à l'origine du groupe**. En effet, dans React Flow, en utilisant la propriété `parentNode` pour les nœuds, leurs coordonnées sont déjà relatives au conteneur. Notre code frontend s'assurera d'assigner `parentNode = id_du_groupe` aux nœuds réarrangés, comme c'est le cas normalement [5]. Le mécanisme existant dans `Whiteboard.tsx` ajustera automatiquement la taille du conteneur groupe pour englober tous les enfants placés (grâce à la fonction `handleNodesChange` qui recalcule la taille d'un groupe en fonction des positions de ses enfants [6] [7]).

En suivant ce pipeline, le backend fournit donc un **service d'organisation sémantique** : on lui envoie les nœuds, il renvoie leur structuration en arbre (positions + relations). Cette séparation permet de maintenir le calcul côté serveur (plus puissant, et gardant la logique métier en un point central).

Emplacement du code dans la structure du projet

Pour intégrer ce nouveau code, voici les endroits clés dans l'arborescence du projet où intervenir :

- **Route API** : créer une nouvelle route dans l'API Next.js. Étant donné que le projet utilise le dossier `app/` de Next.js 13, on placera la route sous `src/app/api/` [1]. Par exemple, créer le dossier `src/app/api/arrange/` et à l'intérieur un fichier `route.ts` (comme illustré plus haut). Ce module exportera la fonction `POST` pour traiter la requête. On peut aussi ajouter `export const runtime = 'nodejs';` en haut si on veut forcer l'exécution côté Node (utile si on utilise des libs CPU-heavy non compatibles Edge).

- **Logique ML de classement** : par propreté, on peut factoriser le code qui fait l'analyse sémantique dans un module utilitaire séparé. Par exemple, créer `src/utils/semanticLayout.ts` ou bien `src/lib/semantic-layout.ts`. Ce module exportera des fonctions comme `findMainNode(nodes): string` (retourne l'id du node principal), `classifyRelations(mainNode, otherNode): 'response' | 'opposition'` (utilisant le

modèle ML ou des heuristiques) et `computeLayout(relations): {positions, edges}` pour calculer les positions. L'endpoint API deviendra plus lisible en appelant ces fonctions utilitaires. **Emplacement** : `src/utils/` est cohérent avec la présence de `src/utils/vector_store.ts` déjà dans le projet [8].

- **Types et modèles de données** : on ajoutera éventuellement des types TypeScript pour clarifier la structure des données échangées. Par exemple, un type `SemanticNode = { id: string, content: string }` pour les nœuds en entrée de l'API, et un type `LayoutResult = { positions: Array<{id: string, x: number, y: number}>, edges: Array<{source: string, target: string, type?: string}> }` pour la sortie. Ces types peuvent être définis dans le module utilitaire ou dans un fichier de types global (selon l'organisation du projet).

- **Intégration base de données** : si on souhaite conserver la structure calculée (par ex. enregistrer les nouvelles positions ou liens en base), on peut modifier la route d'update du projet. On voit qu'il existe une route `PUT /api/projects/[id]` appelée dans `saveNow()` [9]. Notre fonctionnalité pourrait, après calcul du layout, soit :

- retourner le layout au frontend qui l'applique puis laisse l'utilisateur sauvegarder (semi-automatique),
- ou bien écrire directement en base.
  La première approche (frontend applique puis utilisateur enregistre) est plus simple et conforme au fonctionnement actuel (puisque le whiteboard est sauvegardé manuellement ou auto après modifications). On n'a donc pas besoin de modifier la couche Prisma pour stocker immédiatement le résultat ; le `saveNow()` **existant suffira** à persister les nouvelles positions lors de la prochaine sauvegarde utilisateur [10].

En résumé, le code backend se limitera à la nouvelle route API et au module d'IA associé, sans impacter le reste du backend (on réutilise l'authentification existante, le système de tokens d'abonnement, et la structure Prisma des projets). Le code frontend va, quant à lui, interagir avec cet endpoint.

Modèles d'IA légers recommandés pour le ranking/cohérence

Pour identifier les relations sémantiques sans recourir à un LLM coûteux, voici quelques modèles et approches éprouvés, hébergés sur Hugging Face, qui peuvent être utilisés ou fine-tunés :

- **DistilBERT fine-tuné NLI** : *DistilBERT-base-uncased MNLI* (exemple : `typeform/distilbert-base-uncased-mnli`). Ce modèle est entraîné sur l'inférence textuelle (MNLI) et peut servir à déterminer si une phrase *entail* (implique/supporte), *contradict* (s'oppose) ou est *neutral* par rapport à une autre. Il est relativement petit (67M paramètres) et offre une bonne précision (~82% accuracy MNLI [11]). On peut l'utiliser en **zero-shot** pour classer *"réponse vs opposition"* en interprétant *entail/neutral* = *réponse* et *contradiction* = *opposition*. Avantage : largement disponible, licence permissive, déjà en anglais (à utiliser tel quel pour du contenu anglais). Inconvénient : pour du français, il faudrait soit le fine-tuner sur des données FR, soit utiliser un équivalent multilingue.

- **CamemBERT ou DistilCamemBERT (fr)** : Si le contenu des nœuds est majoritairement en français, utiliser un modèle de langue française est pertinent. CamemBERT (6GB) est trop lourd, mais une version distillée existe. On pourrait fine-tuner DistilCamemBERT sur une tâche de *classification de relations* (par exemple, en construisant un petit jeu de données d'entraînement

avec des triplets (texteA, texteB, étiquette) où l'étiquette est "début, réponse ou opposition"). Même avec peu de données, un fine-tuning peut améliorer la cohérence pour notre cas d'usage spécifique. HuggingFace propose des tutoriels de fine-tuning simples pour la classification de paires. Une fois entraîné, on déploie le modèle sur HF Hub pour l'utiliser via l'API.

- **Sentence-Transformers pour la similarité** : En complément de la classification binaire, un modèle d'embeddings sémantiques très léger comme `sentence-transformers/paraphrase-MiniLM-L6-v2` (anglais) ou `sentence-transformers/stsb-xlm-r-multilingual-miniLM` (multilingue) peut être utilisé. Ces modèles produisent des vecteurs 384-D ou 768-D sur lesquels on peut calculer des similitudes cosines. Ils sont rapides et peu gourmands (50–100 Mo). On peut s'en servir pour trouver le nœud central (celui avec le plus de similarité cumulée) et pour regrouper les nœuds par thème. Avantage : pas de classification directe, donc pas besoin de dataset étiqueté, juste des calculs de distances.

- **Modèles de *topic ranking*** : Il existe des modèles spécialisés pour l'appariement question-réponse ou la détection de réponse directe à une question (par ex. `iarfmoose/bert-base-cased-qa-evaluator` sur HF, conçu pour vérifier si une phrase répond à une question [12]). Ce type de modèle peut servir si nos nœuds prennent la forme d'une question et de réponses explicites. Il classera la pertinence d'une phrase B pour répondre à la question A. Cela peut aider à distinguer une *réponse directe* d'un *commentaire tangentiel*. Ce modèle fait ~110M paramètres (base BERT), mais on peut aussi fine-tuner une version plus petite (DistilBERT) sur un jeu Q/R factice.

En pratique, une combinaison **embedding + classification** est sans doute idéale : utiliser l'embedding pour la partie *ranking/cohérence globale* (choix du nœud central, éventuellement ordonner plusieurs réponses par pertinence), et la classification NLI pour la partie *nature de la relation* (soutien vs opposition). Tous les modèles cités sont open-source et certains sont disponibles en version compressée/quantifiée pour une **inférence rapide sur CPU**. De plus, Hugging Face propose une **Inference API** utilisable directement via requête HTTP si l'on veut éviter d'héberger le modèle nous-même (attention aux quotas si on n'a pas de plan payant HF).

Note: Étant donnée la contrainte de ressources (Vercel serverless limite le temps CPU et la mémoire), il peut être judicieux de *précharger* le modèle en mémoire au démarrage de la lambda (en le stockant dans une variable globale du module). Ainsi, sur les invocations à chaud, on ne repaye pas le coût de chargement. On pourrait aussi opter pour un hébergement du modèle sur Aiven ou un micro-service séparé si vraiment nécessaire, mais ce n'est probablement pas justifié pour un modèle léger.

Intégration frontend dans l'interface Whiteboard

Côté frontend, il faut ajouter l'option pour l'utilisateur de déclencher cette réorganisation et appliquer le nouvel agencement dans l'UI React (basée sur React Flow). Voici comment procéder :

- **Bouton "Réarranger" dans l'UI** : On doit décider où placer le déclencheur. Le plus intuitif est d'offrir cette action au niveau d'un **groupe** de nœuds, puisque le réarrangement concerne un groupe à la fois. Deux possibilités ergonomiques :

- *Dans le menu contextuel d'un groupe* – Le projet a déjà un menu contextuel sur les nœuds (clic droit) qui affiche "Demander à l'AI" [13]. On peut y ajouter une nouvelle entrée *"Réorganiser le groupe"* lorsque le nœud cliqué est de type `group`. Concrètement, dans `Whiteboard.tsx`, on

voit que le menu est rendu si `menu` est défini, avec un seul `<button>` actuellement [13]. On peut insérer un second `<button>` juste en dessous, conditionné à `if (getNode(menu.id).type === 'group')`. Par exemple :

```
{menu && (
  <div className="menu-options">
    {getNode(menu.id)?.type === 'group' && (
      <button onClick={() => handleRearrangeGroup(menu.id)}>
        Réorganiser ce groupe
      </button>
    )}
    <button onClick={() => { setAiEditNodeId(menu.id); setMenu(null); }}
>
      <Wand2 size={16} /> Demander à l'IA
    </button>
  </div>
)}
```

Ainsi, sur clic droit d'un groupe, l'option "Réorganiser" apparaît en plus [14]. Sur un nœud non-group, on ne la voit pas. L'icône (double flèche croisée) est optionnelle, mais peut aider à symboliser un arrangement automatique.

- *Dans la barre d'outils/entête* – Le tutoriel d'usage mentionne une barre pour naviguer entre groupes et un bouton de sauvegarde [15] [16]. On pourrait ajouter un bouton "Réarranger" à côté du titre du groupe ou dans la barre d'outils supérieure lorsque l'utilisateur se trouve dans un groupe. Par exemple, si l'UI a un composant pour le titre du projet/groupe (peut-être dans `ClientBoard.tsx` on voit le titre editable [16]), on pourrait mettre un petit bouton à côté du titre du groupe actuel qui, au clic, appelle la fonction de réarrangement. L'avantage est la visibilité directe, l'inconvénient est que ça s'applique au groupe courant uniquement (il faut donc être en "focus" sur le groupe). Sachant que la notion de *focusGroup* existe déjà (`WhiteboardHandle.focusGroup(id)` [17]), on pourrait relier ce bouton au groupe visible.

- **Appel API et mise à jour des positions** : Une fois le bouton cliqué, on implémente la fonction `handleRearrangeGroup(groupId: string)`. Cette fonction va :

- **Récupérer les nœuds du groupe** : on peut utiliser l'état React Flow. Dans `Whiteboard.tsx`, on a la liste `nodes` et la méthode `getNodes()` [18]. Il suffit de filtrer ces nœuds par `parentNode === groupId` (ce qui renvoie les enfants directs du groupe, c.-à-d. les nœuds contenus dans ce groupe). On peut aussi décider d'inclure le nœud group lui-même s'il a un titre pertinent, mais ici on veut surtout les contenus internes.
- **Préparer les données à envoyer** : construire un objet JSON avec seulement l'`id` et le contenu texte de chaque nœud. Par exemple :

```
const groupNodes = getNodes().filter(n => n.parentNode === groupId);
const payload = groupNodes.map(n => ({
  id: n.id,
  content: n.data.text || n.data.label || ''
}));
```

On prend `data.text` si disponible (pour les nœuds texte, markdown, post-it…), sinon `data.label` ou autre champ pertinent. On exclut les nœuds non textuels (s'il y en a dans le groupe) car ils n'ont pas de contenu sémantique pour nous – on pourra quand même les repositionner plus tard soit en les laissant à leur place, soit éventuellement en les traitant à part.

- **Appeler la route API** : via `fetch('/api/arrange', {...})` en méthode POST, en envoyant `JSON.stringify({ nodes: payload })` dans le body, et `Content-Type: application/json`. Cette route retournera le JSON avec les nouvelles positions (et liens). On peut appeler `await fetch` puis `await res.json()` pour obtenir `layoutResult`.
- **Mettre à jour l'état des nœuds** : le résultat contient par exemple `positions: [{id, x, y}, ...]`. On va appliquer ces nouvelles coordonnées aux nœuds concernés. Grâce au state React Flow, on peut soit utiliser `setNodes` pour mettre à jour les positions, soit utiliser l'API imperatively via `reactFlowInstance.setNodes()`. La méthode la plus simple :

```
const { positions, edges } = layoutResult;
setNodes(nodes => nodes.map(node => {
  const newPos = positions.find(p => p.id === node.id);
  if (newPos) {
    return { ...node, position: { x: newPos.x, y: newPos.y } };
  }
  return node;
}));
```

On modifie uniquement les nœuds dont un nouveau positionnement est fourni. Comme on ne change pas la structure (les mêmes nœuds restent enfants du même groupe), React Flow va simplement les redessiner à leur nouvelle place.

Si des **edges** ont été retournés, on peut les intégrer de la même façon : par exemple, faire `setEdges([...edges, ...newEdges])` pour ajouter les nouvelles arêtes aux arêtes existantes (ou remplacer si on préfère ne garder que les auto-liens). Il faut faire attention aux IDs des edges : on peut les générer côté serveur sous forme `${source}-${target}` ou autre, pour éviter les collisions.

- **Rafraîchir l'affichage** : React Flow va automatiquement repositionner les éléments. S'il le faut, on peut appeler `reactFlowInstance.fitView()` pour ajuster le zoom afin que tout le groupe soit visible, surtout si les nœuds se sont beaucoup déplacés. Toutefois, comme on reste dans les limites du conteneur de groupe, ce n'est pas forcément nécessaire. On peut simplement laisser l'utilisateur constater le nouvel arrangement.

Voici à quoi pourrait ressembler `handleRearrangeGroup` dans `Whiteboard.tsx` :

```
async function handleRearrangeGroup(groupId: string) {
  // 1. Récupérer les nœuds du groupe
  const allNodes = getNodes();
  const childNodes = allNodes.filter(n => n.parentNode === groupId);

  // Préparer le payload pour l'API
  const payload = childNodes.map(n => ({
    id: n.id,
```

```
    content: n.data.text || n.data.label || ''
  }));

  // 2. Appel API
  const res = await fetch('/api/arrange', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ nodes: payload })
  });
  if (!res.ok) {
    console.error('API arrange error', await res.text());
    return;
  }
  const { positions, edges: newEdges } = await res.json();

  // 3. Mettre à jour les positions des nœuds dans l'état
  setNodes(nodes => nodes.map(node => {
    const updated = positions.find(p => p.id === node.id);
    return updated
      ? { ...node, position: { x: updated.x, y: updated.y } }
      : node;
  }));
  // 4. Ajouter les nouvelles arêtes (liens) si fournis
  if (newEdges?.length) {
    setEdges(eds => [...eds, ...newEdges]);
  }

  // 5. (Optionnel) Ajuster la vue pour voir tout le groupe
  reactFlow.fitView({ nodes: [{ id: groupId }], padding: 0.2 });
}
```

Il faudra penser à **fermer le menu contextuel** après clic (comme ils le font avec `setMenu(null)`). Dans l'exemple ci-dessus, si on appelle `handleRearrangeGroup` depuis le bouton contextuel, on pourrait appeler `setMenu(null)` juste après pour cacher le menu.

- **UX** : on peut afficher un léger indicateur de chargement pendant la requête (par ex, désactiver le bouton ou afficher un spinner sur le groupe). Vu que le calcul est assez rapide (<1s généralement), ce n'est pas obligatoire, mais utile si la latence est perceptible. On peut par exemple utiliser un état `isArranging` dans Whiteboard, et si vrai, griser l'écran ou montrer un message "Réorganisation en cours…".

- **Application des nouvelles coordonnées** : grâce au code ci-dessus, les nœuds se déplacent instantanément. Le `GroupNode` va automatiquement redimensionner son cadre via le code existant [19] [20]. Ainsi, si les nœuds s'étendent plus loin qu'avant, le conteneur grandira pour les inclure (avec une marge de 16px définie dans `padding`). Si au contraire ils sont maintenant plus concentrés qu'avant, le conteneur **ne rétrécira pas automatiquement** (car la logique actuelle ne réduit pas la taille, elle fait seulement grandir si nécessaire). Ce détail signifie que si un groupe était très étendu puis qu'on compacte les nœuds, le fond du groupe peut rester de grande taille initialement. Ce n'est pas critique visuellement, mais éventuellement, on pourrait affiner `handleNodesChange` pour réduire la taille si beaucoup d'espace vide. Cela peut être

une amélioration ultérieure (par ex, recalculer needW/needH à la baisse si les enfants laissent une marge vide importante).

- **Sauvegarde** : Une fois satisfait du nouvel arrangement, l'utilisateur peut cliquer sur *Sauvegarder* (bouton déjà présent dans l'UI [21] qui déclenche `whiteboardRef.current.saveNow()` [21]). La sauvegarde enverra au backend le projet avec les nouvelles positions, via `PUT /api/projects/{id}` [9]. Ainsi, la persistance est assurée. On peut aussi envisager d'appeler automatiquement `saveNow()` à la fin de `handleRearrangeGroup` pour plus de commodité, mais c'est au choix (certains utilisateurs préfèreront vérifier le résultat avant de sauvegarder). Dans un premier temps, rester cohérent avec le mode manuel est bien.

En termes d'**organisation du code frontend**, `ClientBoard.tsx` importe et rend le composant `Whiteboard` [22]. La fonction de réarrangement peut être définie soit *à l'intérieur* de Whiteboard (puis exposée via une ref ou via props), soit au niveau de ClientBoard si on a accès aux nodes. Cependant, comme toute la logique de manipuler les nodes est déjà dans `Whiteboard.tsx`, il est plus simple d'implémenter `handleRearrangeGroup` dans ce composant (aux côtés de fonctions comme `handleAiNodeSubmit`, etc.). On veillera à **passer l'ID du groupe** correctement. Dans le menu contextuel, on a `menu.id` qui correspond à l'ID du nœud sur lequel on a fait clic-droit [23]. Donc dans le cas d'un groupe, `menu.id` est l'ID du group-node lui-même, ce qui est exactement ce qu'il nous faut pour filtrer ses enfants.

Enfin, on ajoutera peut-être une petite **indication visuelle** pour l'utilisateur : par exemple un tooltip "Réorganiser ce groupe selon les relations sémantiques" quand on survole le bouton, afin d'expliquer la fonctionnalité.

Conseils d'infrastructure (Vercel, Firebase Auth, Aiven, Stripe)

Intégrer cette fonctionnalité de manière fluide dans l'infrastructure existante nécessite de prendre en compte :

- **Déploiement sur Vercel** : La route API Next.js fonctionnera sur les fonctions serverless de Vercel. Il faut faire attention aux temps de réponse et à la taille des dépendances :
- *Durée d'exécution* : l'analyse de ~10 nœuds via un petit modèle devrait prendre < 1 seconde sur CPU, ce qui est largement acceptable dans la limite (~10s max) de Vercel Serverless. Si on utilise l'API HuggingFace distante, la latence réseau s'ajoute – mais là encore quelques secondes au pire. Nous restons donc dans les clous.
- *Taille des bundles* : importer `transformers` ou un modèle local peut alourdir le déploiement. Une astuce pour minimiser cela est d'utiliser uniquement ce qui est nécessaire (par ex, on pourrait utiliser `@xenova/transformers` qui charge des poids ONNX en streaming, ou charger un modèle via CDN). Si la taille du lambda dépasse ~50 MB, Vercel pourrait poser problème. En ce sens, utiliser l'**Inference API** (HTTP) de HuggingFace est un compromis : pas de poids de modèle dans le bundle, mais dépendance réseau. Alternativement, on peut héberger le modèle distillé nous-même sur Aiven (voir ci-dessous).

- *Edge Functions* : pour des temps de réponse encore moindres, Next 13 permet de déployer des routes API en tant que *Edge Middleware* (basées sur V8). Cependant, toutes les libs ML ne tournent pas en environnement Edge. Une option serait d'utiliser un modèle TensorFlow.js ou ONNX et de l'exécuter dans le contexte Edge. Vu la complexité, on peut se contenter du Node.js runtime (par défaut) [24] pour cette route.

- **Firebase Auth** : le projet utilise Firebase Auth (probablement via le cookie `uid` et la lib `auth.ts` [25]). Notre endpoint doit respecter la même protection. Comme montré, on récupère `uid` via `cookies()` [1] et on vérifie l'utilisateur en base. Il convient de s'assurer que le cookie `uid` est bien transmis lors de l'appel `fetch('/api/arrange')`. Par défaut, les requêtes same-origin en Next utilisent les cookies du navigateur, donc ça devrait être transparent. Si besoin, on peut ajouter `{ credentials: 'include' }` au fetch pour être certain. En sortie, si l'API retourne un 401 ou 403 (non autorisé ou quota dépassé), on pourrait gérer ça côté UI (afficher un message "Connectez-vous" ou "Limite atteinte"). Le code d'exemple plus haut renvoie des JSON d'erreur en cas de problème d'auth/tokens [26], donc le frontend peut les utiliser.

- **Aiven (PostgreSQL)** : Aiven héberge la base PostgreSQL du projet. La persistance des nœuds et projets y est sans doute gérée via Prisma (comme on voit l'utilisation de `prisma.user` et sûrement `prisma.project` ailleurs). Notre feature n'ajoute pas de nouvelles tables, elle utilise les champs existants:

- Les positions *x*, *y* de chaque nœud sont probablement déjà stockées (peut-être dans un champ JSON ou dans une table dédiée aux nœuds). En sauvegardant le projet après réarrangement, ces coordonnées mettront à jour la base. Aucune migration n'est nécessaire.
- Si on décide d'enregistrer des *liens sémantiques* (edges), et que la structure actuelle du projet ne les stocke pas, il faudrait prévoir où le faire. Possiblement, le projet n'enregistrait pas les edges (puisque c'était un whiteboard libre). Si on veut les garder, on peut ajouter un champ `edges` (JSON) dans la table project ou dans la structure du champ `nodes`. Dans un MVP, on peut aussi ne pas sauvegarder les edges auto-générés (ils peuvent être recréés à la demande).
  Recommandation : enregistrer les edges aussi, pour que l'utilisateur retrouve le visuel complet à la reconnexion. Vu que `saveNow()` envoie `edges` aussi [27], il y a de fortes chances que les edges soient déjà persistés (peut-être dans un champ JSONB). Vérifions vite : dans `Whiteboard.tsx`, initialisation `const defaultInitialEdges: Edge[] = []` [28] et plus loin `fetch('/api/projects/'+id, {method: 'PUT', body: JSON.stringify({ title, nodes, edges })})` [29] montre qu'on enregistre bien les edges. Donc tout va bien : nos edges générées seront sauvegardées comme les autres.

En termes de charge sur la base, cette feature n'en ajoute pas de significative : on fait juste un `findUnique` user et un `update` user (pour tokens) dans l'endpoint [30] [31], ce qui est négligeable. L'enregistrement du projet se fait déjà.

- **Stripe (abonnements)** : Le code montre une gestion de `plan.aiTokens` et `aiTokensUsed` [2]. Cela signifie que l'usage de l'IA est limité selon l'abonnement (géré via Stripe probablement). Il faut décider comment compter l'utilisation de cette nouvelle fonctionnalité. Deux choix :

- **Consommer des jetons AI existants** : On peut assimiler le *"réarrangement sémantique"* à une action AI et décrémenter le compteur de jetons de l'utilisateur. Par exemple, on pourrait estimer un coût fixe (ex: 50 tokens par réorganisation) ou un coût variable (ex: 1 token par nœud analysé). Une implémentation simple : après avoir identifié les relations, incrémenter `user.aiTokensUsed` d'un certain montant (via `prisma.user.update` comme fait dans `onFinish` de la requête streaming [32]). On peut même faire l'incrément *avant* de renvoyer la réponse, ou réutiliser `onFinish` si on intègre notre appel ML dans `streamText`. Cependant, notre pipeline n'utilise pas `streamText` (qui sert aux réponses en streaming de l'LLM). Il faudra donc gérer manuellement. Ex:

```
const cost = Math.max(1, nodes.length - 1); // par ex, 1 token par
comparaison
await prisma.user.update({ where: { id: user.id }, data: {
aiTokensUsed: { increment: cost } } });
```

Ce bloc serait inséré avant de renvoyer la réponse JSON. **NB** : Veillez à utiliser une transaction ou bien à tolérer une petite imprécision si la requête coupe avant d'incrémenter (peu probable ici). Vu la faible consommation relative, on peut aussi choisir de ne pas compter cette action dans les jetons afin de la laisser disponible même aux offres gratuites (argument : c'est léger, ça incite à utiliser l'outil). Si on décide cela, il suffit de ne pas ajouter de consommation de tokens dans notre route (et éventuellement de ne pas faire de check `plan.aiTokens===0` pour cette route). Néanmoins, pour la cohérence, mieux vaut respecter les mêmes conditions d'accès que pour le nœud AI : on peut réutiliser exactement le même contrôle d'abonnement [2], ainsi seuls les utilisateurs autorisés à l'IA y auront accès (ce qui est logique car c'est une fonction AI).

- **Ne pas consommer de jetons** : Si on estime que la fonctionnalité est un bonus "mineur" en ressources (ce qui est vrai comparé à un appel GPT-4), on peut décider de la fournir librement à tous les utilisateurs connectés. Dans ce cas, on peut ignorer/désactiver la partie comptage pour cette route. Par exemple, on pourrait retirer ou ajuster les conditions dans le code de l'API : autoriser même si `plan.aiTokens === 0`. **Solution propre** : définir un *flag* dans l'abonnement, du genre `plan.allowSemanticArrange` qui serait vrai dès le plan gratuit. Mais plus simplement, on peut considérer que `plan.aiTokens===0` signifie "pas d'LLM", mais pas forcément pas d'IA du tout. Ce point est business : pour notre implémentation technique, on suivra la règle donnée par l'équipe produit.

- **Monitoring et logs** : Sur Vercel, on peut ajouter des logs (`console.log`) pour monitorer l'usage de la feature. Par exemple, logguer `user.id` et le nombre de nœuds traités, le temps de calcul, etc. Cela aidera à ajuster si besoin (par ex. si on constate des temps anormaux pour certains inputs). Éventuellement, on peut ajouter une traçabilité via Analytics front (il y a un hook `useAnalytics` et des appels `track()` dans le code [33]). On pourrait faire `track('arrange_group', { count: nodes.length, userId: userStatus.id })` dans `handleRearrangeGroup` pour alimenter Mixpanel ou autre, afin de voir la popularité de la fonctionnalité.

En conclusion, notre plan s'insère bien dans l'architecture existante : **Vercel** hébergera la logique ML légère sans problème, **Firebase Auth** sécurisera l'accès comme pour les autres features AI, **Aiven** stockera simplement les résultats via la sauvegarde normale des projets, et **Stripe** pourra, en fonction des règles décidées, contrôler l'accès ou la consommation liée à cette fonctionnalité. En suivant ces bonnes pratiques (modularité du code, appels réseau optimisés, respect des quotas utilisateurs), la fonctionnalité de réarrangement automatique sémantique pourra être déployée de manière robuste et évolutive au sein du whiteboard .

---

1 2 24 26 30 31 32 route.ts
https://github.com/HocineBoudieb/white_board/blob/7886de77d4c337f0e5d06fb2c304475df2ac5485/src/app/api/groq/route.ts

3 rapport_whiteboard.tex
https://github.com/HocineBoudieb/white_board/blob/7886de77d4c337f0e5d06fb2c304475df2ac5485/docs/rapport_whiteboard.tex

4 11 typeform/distilbert-base-uncased-mnli · Hugging Face
https://huggingface.co/typeform/distilbert-base-uncased-mnli

5 6 7 8 9 10 13 14 17 18 19 20 23 27 28 29 Whiteboard.tsx
https://github.com/HocineBoudieb/white_board/blob/7886de77d4c337f0e5d06fb2c304475df2ac5485/src/components/Whiteboard.tsx

12 iarfmoose/bert-base-cased-qa-evaluator - Hugging Face
https://huggingface.co/iarfmoose/bert-base-cased-qa-evaluator

15 16 21 22 33 ClientBoard.tsx
https://github.com/HocineBoudieb/white_board/blob/7886de77d4c337f0e5d06fb2c304475df2ac5485/src/app/projects/[id]/ClientBoard.tsx

25 layout.tsx
https://github.com/HocineBoudieb/white_board/blob/7886de77d4c337f0e5d06fb2c304475df2ac5485/src/app/layout.tsx