Super — l'environnement est prêt. La bonne stratégie maintenant, c'est **socle → refactors mécaniques → import progressif "slice par slice" → ajustements manuels**. Donc: **ne commence pas par des refactors manuels IntelliJ**. Branche d'abord les outils (OpenRewrite / Error Prone / Semgrep), fais passer les "migrations 1:1" automatiques, puis seulement ensuite touche au code métier.

**Méthodo proposée (ordre conseillé)**

1. **Valider le socle Forge "vierge"**
    - MDK 1.20.1 installé, genIntellijRuns, classe principale @Mod("minegicka"), mods.toml et gradle.properties cohérents (MODID/group/licence). Lance runClient pour vérifier que le mod vide démarre.
2. **Brancher les outils de refactor/qualité** (avant d'importer le legacy)
    - OpenRewrite (plugin + rewrite.yml avec recettes mécaniques — ex. cpw.mods.fml → net.minecraftforge.fml, ResourceLocation package), Error Prone activé, et Semgrep avec règles de détection (IEEP, SimpleNetworkWrapper). Exécute rewriteRun et une compilation à vide pour valider la chaîne.
3. **Auditer le projet legacy (lecture froide)**
    - Greps ciblés pour repérer les API 1.7.10 à migrer (registries, IEEP, ancien netcode, IIcon/TESR, worldgen…), et optionnellement un montage "compilable" via RFG pour obtenir des erreurs/points durs concrets. Résultat attendu: **carte des migrations** (DeferredRegister, Capabilities, netcode moderne, Datagen + modèles JSON).
4. **Importer très petit, "canari", et automatiser**
    - Copie **de quelques classes utilitaires**/constantes depuis le legacy dans le projet 1.20.1, puis rewriteRun → compile. Tu corriges les imports/types mécaniques restants. Objectif: prouver que le pipeline "import → rewrite → compile" est sain avant de tout ramener.
5. **Installer les garde-fous de cohérence**
    - Vérifie l'alignement **MODID / group / package / mods.toml**, et privilégie ResourceLocation.tryParse pour éviter les warnings. Mets en place .editorconfig (et éventuellement Spotless) pour stabiliser le formatage.
6. **Premier "vertical slice" (ex: un item simple)**
    - Import **d'un seul feature minimal** (p.ex. 1 item): code + DeferredRegister, DataGen (runData) pour générer les JSON (models/recipes/loot/tags), puis runClient pour le voir en jeu. Ce slice te force à exercer **registries + ressources + datagen** sans la dette du reste.
7. **Écrire/améliorer les recettes OpenRewrite/Refaster**
    - Quand un pattern de migration est stable (ex. GameRegistry.register → DeferredRegister), **consolide-le en recette** pour le rejouer à grande échelle. C'est là que tu capitalises pour les prochains imports massifs.

8. **Réseau & données joueur**
    - Remplacer IEEP par **Capabilities** et migrer l'ancien réseau (détecté par Semgrep) vers l'implémentation moderne. Procède par **adaptateurs minces** d'abord, puis nettoyage. Compile, et couvre avec un **GameTest** quand c'est possible (runGameTestServer).
9. **Rendu/UI & logique serveur**
    - Traiter ce qui est client-side via runClient, et la logique serveur/monde via runServer pour éviter les cross-side calls. Un seul run à la fois, Gradle daemon actif.
10. **Itérer slice par slice jusqu'au cœur du mod**
    - Monde/génération, entités, blocs complexes, équilibrage & configs. À chaque slice: **Definition of Done** = compile OK, DataGen commitée, test en jeu (runClient) + si possible GameTest vert, style/formatage propre.

---

**Pourquoi éviter le refactor manuel IntelliJ au début ?**

- Les **migrations mécaniques** (packages, types renommés, API 1:1) sont **plus sûres et reproductibles** via OpenRewrite/Refaster que via clics manuels. Tu les rejoues à chaque vague d'import.
- IntelliJ reste excellent pour **refactors structurels ciblés** (SSR, safe-delete, rename) **après** que les passes automatiques aient réduit le bruit.

---

**Plan de fractionnement (backlog opérationnel)**

- **Pré-import (socle)**
    - Forge "vierge" + runs (genIntellijRuns, runClient) ✓
    - .editorconfig (+ Spotless optionnel) ✓
    - OpenRewrite / Error Prone / Semgrep branchés ✓
- **Audit legacy**
    - Greps API 1.7.10 → feuille de route de migration
    - (Option) Build RFG pour lister les points durs concrets
- **Import canari**
    - Petite poche de code → rewriteRun → compileJava
- **Slices fonctionnels** (boucle)

1. Item/Bloc simple (registries + datagen) → test en jeu
2. Capabilities joueur + persistance

3. Réseau (paquets essentiels)
4. Rendu & interactions
5. Worldgen/serveur
6. Polissage (assets, localisation, recettes)
    - À chaque slice: runData, runClient/runServer, **GameTests** si pertinents.

---

**Commandes utiles (rappel)**

- ./gradlew genIntellijRuns, runClient, runServer, runData, runGameTestServer, rewriteRun, clean compileJava. Garde **un seul run à la fois**.

---

Si tu veux, je peux te proposer un **premier "vertical slice" concret** (classes à créer, registries, stub de Capability, et check-list DataGen) adapté à Minegicka3, en partant de l'item le plus simple du mod.