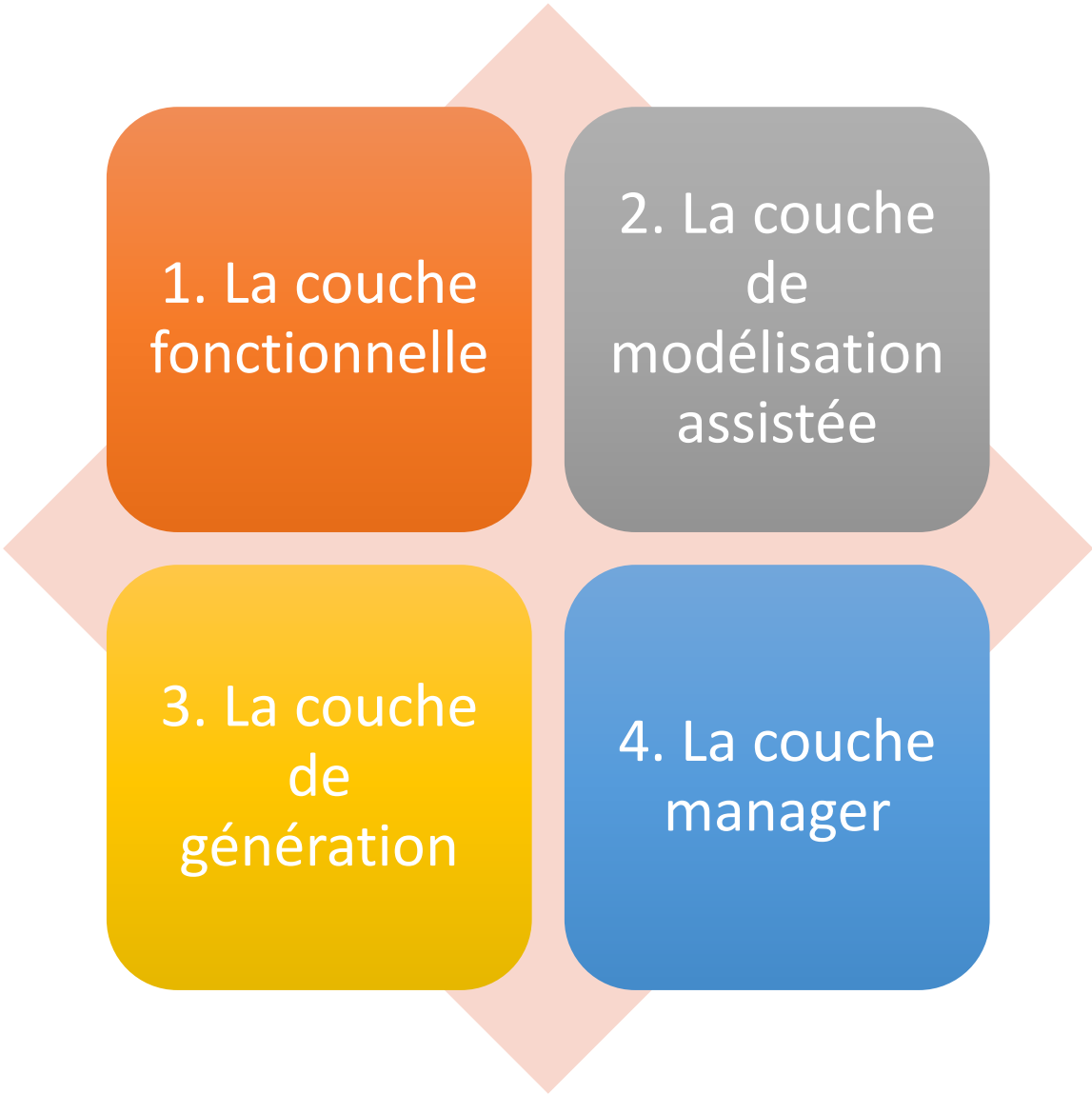
DATAMODEL MANAGER

BY IDEO-LAB

APRIL 2026

GUILLAUME ONEILL – LOCKHEED LLC

AGENDA


1. La couche fonctionnelle
2. La couche de modélisation assistée
3. La couche de génération
4. La couche manager

LES BASES DU PRODUIT

1. La couche fonctionnelle

L'utilisateur décrit son besoin métier de manière simple :

- "Je veux une table Client"
- "Un client peut avoir plusieurs commandes"
- "Une commande appartient à un seul client"
- "Je veux stocker une date, un montant, un statut"
- "Ce champ doit être obligatoire"
- "Ce champ doit être unique"
- "Je veux une relation many-to-many avec Produit"

Donc on n'impose pas immédiatement la syntaxe Django.

On part d'un langage **simple, humain, guidé, pédagogique**.

2. La couche de modélisation assistée

Par exemple :

- si l'utilisateur crée `Order` puis `Customer`, le système peut proposer automatiquement :
  - `customer = models.ForeignKey(...)`
- s'il crée `Product` et `Category` avec relation multiple, il propose :
  - `ManyToManyField`
- s'il met un champ "email", le système peut suggérer :
  - `EmailField`
- s'il met un champ "description longue", il peut suggérer :
  - `TextField`
- s'il met "created at", il peut suggérer :
  - `DateTimeField(auto_now_add=True)`

Autrement dit, le système joue le rôle de **traducteur entre besoin métier et vraie structure Django**.

L'outil transforme ces inputs en une **représentation structurée du data model** :

- entités
- champs
- types
- relations
- contraintes
- options Django
- commentaires métier
- commentaires techniques
- suggestions automatiques

C'est ici que ton système devient intelligent.

3. La couche de génération

3. La couche de génération

Une fois le modèle validé, l'outil génère :

- le vrai `models.py`
- éventuellement les `admin.py`
- éventuellement les serializers
- éventuellement les forms
- éventuellement les vues CRUD
- éventuellement un export JSON/YAML intermédiaire
- éventuellement les migrations initiales
- et surtout un **diagramme du data model**

Donc on a deux sorties :

LES SORTIES ATTENDUES

Sortie A – Code Django réel

```
class Customer(models.Model):
    full_name = models.CharField(max_length=255)
    email = models.EmailField(unique=True)
    created_at = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.full_name
```

LES SORTIES ATTENDUES

- Sortie B - Diagramme Lisible

sortie B — diagramme lisible

Sous forme :

- ERD simplifié
- diagramme orienté Django
- vue métier
- vue technique

4. La couche manager

Et là, à mon avis, c'est la partie la plus forte de ton idée.

Parce qu'un générateur seul, c'est bien.
Mais un **manager de data models**, c'est beaucoup plus puissant.

Cela veut dire que le système peut :

- sauvegarder les modèles créés
- versionner les changements
- comparer deux versions
- détecter les impacts
- regénérer le code
- regénérer le diagramme
- gérer plusieurs projets
- gérer plusieurs apps Django
- documenter automatiquement le modèle

Donc on ne parle plus d'un simple assistant.
On parle d'un **atelier de conception de schéma applicatif**.

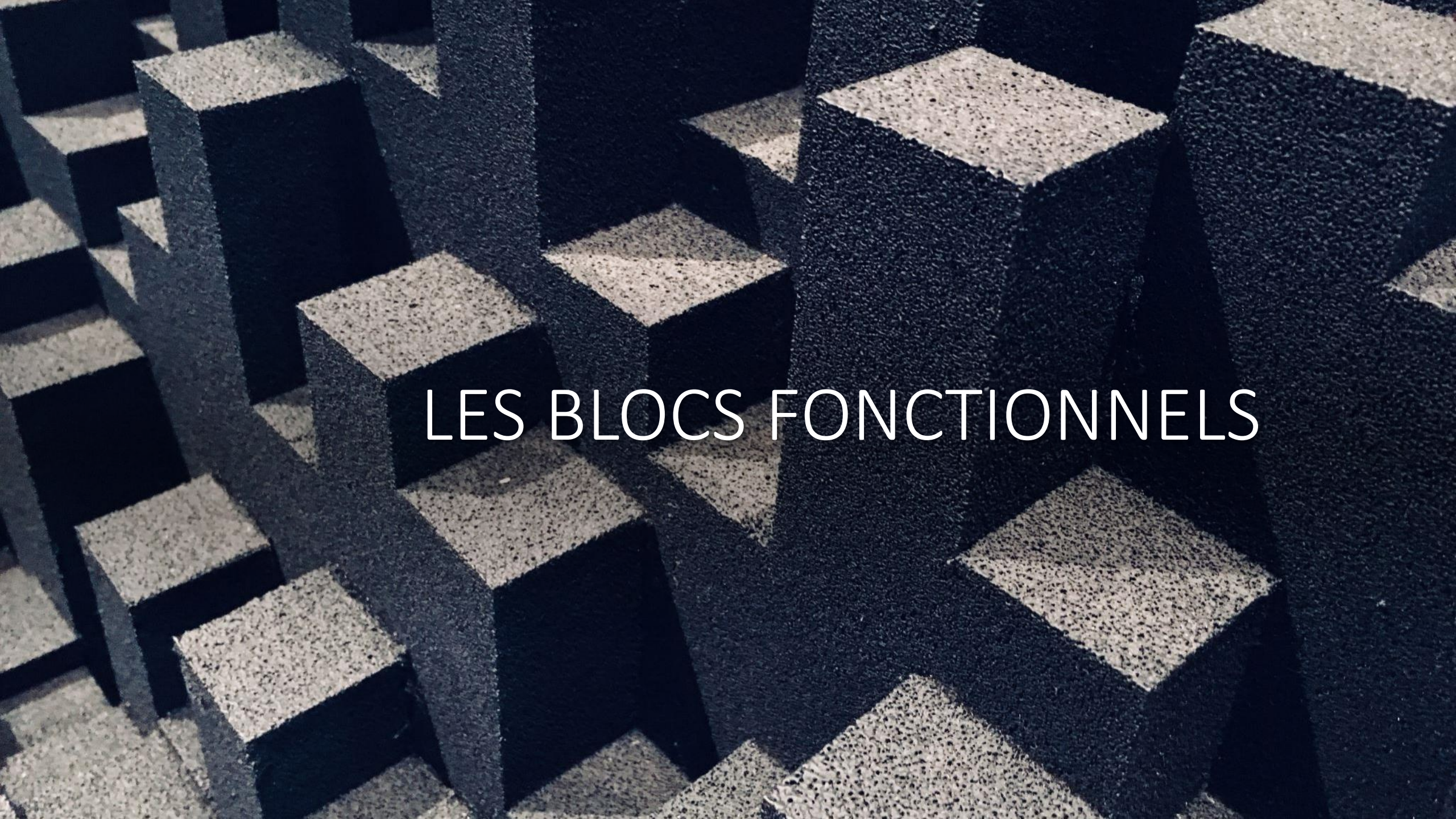
LES BUTS POURSUIVIS

- le code propre
- les explications
- les relations correctes
- le diagramme
- la documentation

- des devs juniors
- des devs non spécialistes Django
- des analysts fonctionnels
- des architectes
- des chefs de projet techniques
- voire des clients avancés

La bonne philosophie produit

- Niveau 1 — métier
- Niveau 2 — traduction technique guidée
- Niveau 3 — génération Django experte



LES BLOCS FONCTIONNELS

Les grands blocs fonctionnels du futur produit

- A. Project / App manager
- B. Entity builder
- C. Field builder
- D. Relationship builder
- E. Code preview engine
- F. Diagram engine
- G. Validator / advisor
- H. Generator engine
- I. Versioning / compare

A. Project / App manager

Il faut pouvoir gérer :

- plusieurs projets
- plusieurs apps Django
- plusieurs modèles par app

Exemple :

Projet `crm_platform`

App `customers`

App `orders`

App `billing`

B. Entity builder

C'est le coeur de l'outil.

L'utilisateur crée une entité :

- nom technique
- nom lisible
- description
- commentaire métier
- commentaire technique

Exemple :

- Technical name: `Customer`
- Display name: `Client`
- Description: "Represents a customer account in the CRM"

C. Field builder

Pour chaque modèle, il ajoute des champs via un gros formulaire intelligent.

Chaque champ aurait :

- field name
- field label
- field type
- required
- unique
- indexed
- default value
- help text
- max_length
- null / blank
- choices
- comment
- example value

Et surtout :
le formulaire doit s'adapter selon le type.

D. Relationship builder

Bloc dédié aux relations :

- OneToOne
- OneToMany
- ManyToMany
- self relation
- optional relation
- reverse name
- relation comment

Le système doit être ultra pédagogique ici, parce que c'est souvent là que les non-experts se perdent.

on demande simplement :

- source model : `Order`
- target model : `Customer`
- relation type : “many orders belong to one customer”
- relation required : yes/no
- delete behavior : cascade / protect / set null
- reverse name : `orders`

Puis le moteur traduit.

E. Code preview engine

À droite de l'écran, tu affiches en temps réel :

- preview `models.py`
- preview commentaire explicatif
- preview pseudo-SQL
- preview diagramme

Ça, c'est très fort UX.

F. Diagram engine

Tu génères un diagramme dynamique.

Je te conseille 3 modes :

mode 1 — simple

Boîtes avec noms de modèles + champs + relations

mode 2 — fonctionnel

Vue métier plus lisible pour un non-dev

mode 3 — Django/tech

Avec détails techniques :

- type
- null
- unique
- FK/M2M
- indexes éventuels

G. Validator / advisor

Le système doit vérifier la cohérence.

Exemples :

- champ `email` en `CharField` → suggérer `EmailField`
- `DecimalField` sans `max_digits` → erreur
- FK nullable sans `on_delete=SET_NULL` → alerte possible
- `ManyToManyField` sur un besoin qui ressemble à une table métier → suggérer un `through model`
- champ `name` sans `max_length` → blocage
- relation ambiguë entre deux modèles → suggestion de clarification

H. Generator engine

Il produit les fichiers :

- `models.py`
- `admin.py`
- `apps.py`
- éventuellement `serializers.py`
- éventuellement `forms.py`
- éventuellement `views.py`
- éventuellement `urls.py`
- documentation markdown/html
- export diagram PNG/SVG
- export JSON de configuration interne

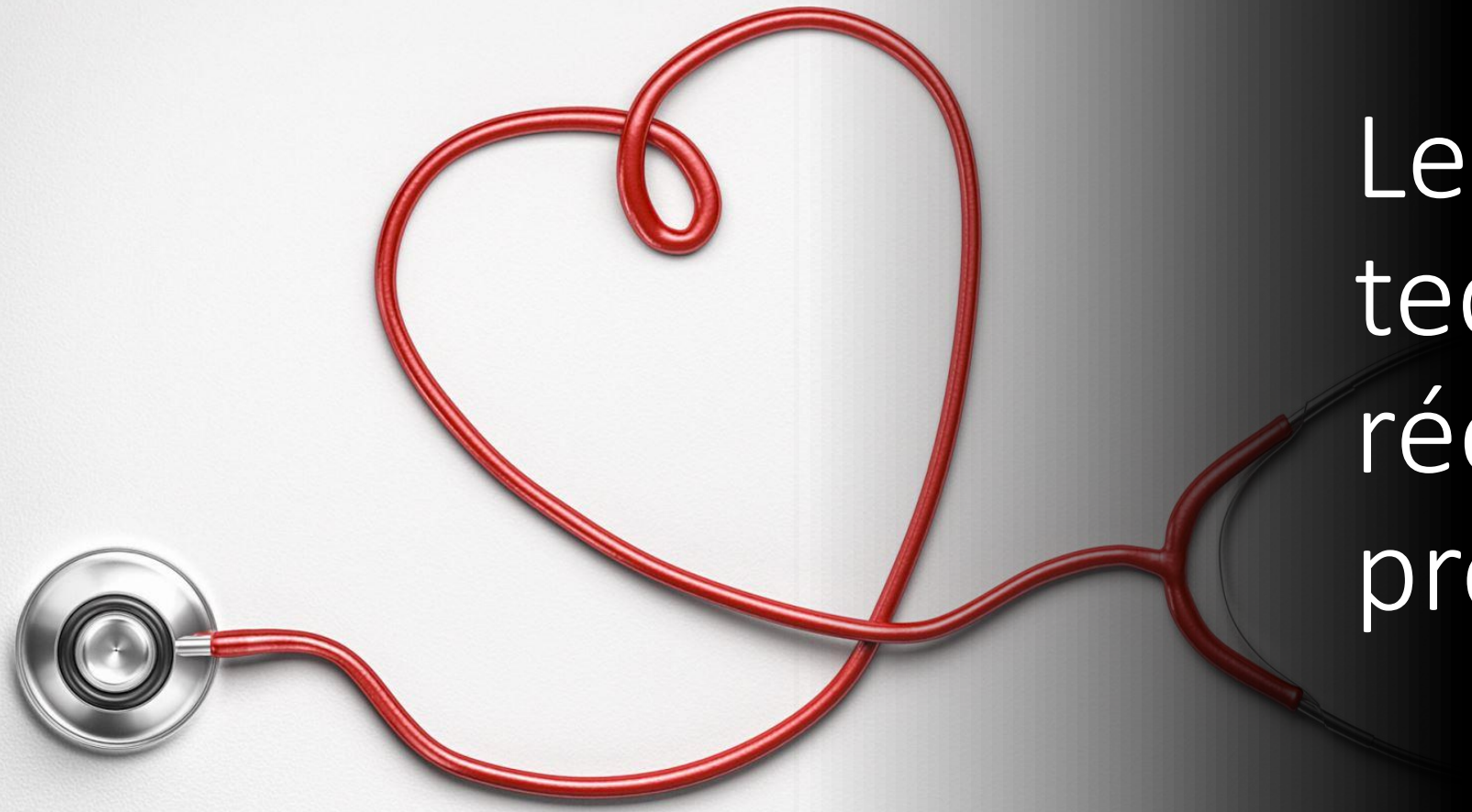
I. Versioning / compare

Très important.

Tu dois pouvoir :

- sauvegarder version 1
- modifier le modèle
- voir les différences
- regénérer
- potentiellement préparer les futures migrations

Là, tu entres dans une logique quasi “schema designer”.



Le coeur technique réel du produit

AGENDA TECHNIQUE DU PRODUIT

- 1. Le data model de conception
- 2. Le data model généré